www.prolog-heritage.org

Version restaurée à partir des archives de PrologIA gracieusement cédées à l'association PROLOG HERITAGE. Manuel mis en consultation libre sur Internet au printemps 2010.
© PROLOG HERITAGE

Contact : association@prolog-heritage.org

Association Prolog
HERITAGE

Prolog III

Version 1.3

Décembre 1991

© PrologIA

Association Prolog
HERITAGE

Garantie et responsabilités

PrologIA n'offre aucune garantie, expresse ou tacite, concernant ce manuel ou le logiciel qui y est décrit, ses qualités, ses performances ou sa capacité à satisfaire à quelque application que ce soit.

PrologIA ne pourra être tenue responsable des préjudices directs ou indirects, de quelque nature que ce soit, résultant d'une imperfection dans le programme ou le manuel, même si elle a été avisée de la possibilité que de tels préjudices se produisent. En particulier, elle ne pourra encourir aucune responsabilité du fait des données mémorisées ou exploitées, y compris pour les coûts de récupération ou de reproduction de ces données.

L'acheteur a toutefois droit à la garantie légale dans les cas et dans la mesure seulement où la garantie légale est applicable nonobstant toute exclusion ou limitation.

Droits d'auteur

Ce manuel et le logiciel qu'il décrit sont protégés par les droits d'auteur. Au terme de la législation traitant de ces droits, ce manuel et ce logiciel ne peuvent être copiés ou adaptés, en tout ou en partie, sans le consentement écrit de PrologIA, sauf dans le cadre d'une utilisation normale ou pour faire une copie de sauvegarde. Ces exceptions n'autorisent cependant pas la confection de copies à l'intention d'un tiers, que ce soit ou non pour les vendre.

Prolog III est une marque déposée de PrologIA.

Préambule

Félicitations ! Vous avez entre les mains le leader d'une nouvelle génération de langages de programmation en logique : Prolog III. Ce langage s'appuie sur le concept nouveau de programmation logique par contraintes. Si on y retrouve, bien sûr, les principes généraux qui ont fait de Prolog le langage de programmation phare de l'Intelligence Artificielle, Prolog III représente surtout une refonte totale du cœur même du langage, l'unification. Ceci se traduit par une manipulation plus fine des arbres, finis et infinis, qui demeurent les objets principaux utilisés par le langage, mais également par la généralisation de la notion fondamentale, et seulement ébauchée dans Prolog II, de résolution de contraintes. C'est grâce à cette notion que l'on peut, en Prolog III, dégager une approche du traitement numérique et du traitement de l'algèbre de Boole fondamentalement différente des essais qui ont été tentés dans ces domaines par l'adjonction de prédicats évaluables multiples et fort éloignés de la philosophie de base. Prolog III permet donc de poser des systèmes d'équations et inéquations portant sur des nombres réels ou rationnels en précision parfaite, de traiter des formules en logique propositionnelle complète, et de résoudre un certain nombre de contraintes portant sur les arbres et les listes.

Ce remaniement profond, et la puissance qui en découle, ne sont cependant pas gratuits. Le cœur de l'interprète est cinquante fois plus gros que celui de Prolog II, et le pari de s'attaquer à des problèmes exponentiels, comme le traitement de l'algèbre de Boole ou la simplification de systèmes d'équations linéaires, amène parfois des déconvenues du point de vue des temps d'exécutions. Il faut donc rester conscient du fait que la puissance d'expression du langage voile parfois la complexité de la résolution qui en découle.

© PrologIA

Terminons ce premier contact en espérant que vous apprécierez la programmation en Prolog III, et que vous serez aidés en cela par ce manuel. Nous avons tenté d'y décrire, mais surtout de commenter et d'expliquer les nombreuses notions nouvelles qui font de Prolog III un langage qui se démarque fondamentalement des autres Prolog. Ce souci de produire un discours à la fois précis dans les fonctionnalités et didactique dans les concepts nous conduit, de fait, à considérer ce manuel comme une première étape. Et il nous apparaît évident que seuls votre aide, vos remarques et vos critiques nous permettrons de réellement faire évoluer cette documentation vers un outil dont l'ambition est de vous amener à pénétrer jusque dans ses moindres détails un langage qui initialise une nouvelle étape de la programmation logique.

En attendant vos critiques et remarques, nous vous souhaitons beaucoup de satisfactions avec Prolog III.

Les auteurs

© PrologIA

Différences entre la version 1.2 et la version 1.3

Cette nouvelle version de Prolog III apporte un certain nombre de modifications par rapport à la version précédente. Bien sûr, les bogues connus ont été corrigés, et certaines lacunes de la documentation comblées. Nous remercions à ce propos tous ceux qui ont eu la gentillesse de nous faire part de ces imperfections. Mais l'objectif essentiel de cette refacture est l'implantation d'un "débogueur" symbolique que nous n'hésiterons pas à appeler *debugger* dans tout le manuel !

Cette nouvelle version voit également l'introduction de nouvelles primitives numériques.

Il est maintenant possible de récupérer et traiter les erreurs se produisant en cours d'exécution, par l'extension des primitives *block* et *block_exit*.

En ce qui concerne les règles prédéfinies et procédures externes, nous en avons ajouté de nouvelles, ou parfois mieux documenté d'anciennes. Voici la liste de ces primitives, qui sont décrites en détail dans le chapitre réservé à cet effet.

Ont été ajoutées ou étendues :

```
'!'
block(...)
block_exit(...)
break(S1)
break(S1,S2)
debug
lower_bound(R1,R2)
maximize(R)
maximum(R1,R2)
minimize(R)
minimum(R1,R2)
no_debug
no_spy(<I,A>)
no_spy(S)
set_config(S,N)
spy(<I,A>)
spy(S)
upper_bound(R1,R2)
```

© PrologIA

Des primitives numériques plus performantes pour l'optimisation de problèmes linéaires (**`minimize`**, **`maximize`**) sont introduites.

Les primitives **`min_value`**, **`max_value`** sont désormais obsolètes[1] et remplacées par **`minimum`**, **`lower_bound`**, **`maximum`**, **`upper_bound`** .

Le prédicat **`cut`** a été retiré de cette version : **`'!'`** doit être utilisé à la place.

De nouveaux messages d'erreur ont été introduits, dont en particulier "Fin de fichier", et "Fin de fichier inattendue". Ceci afin de permettre une meilleure prise en compte des erreurs de lecture.

Signalons au programmeur que, par défaut, la tentative d'effacement d'une règle non-définie ou d'un but non-exécutable affiche un avertissement sur la console. L'option *"undefined_rule"* du prédicat *set_config* est maintenant positionnée à *"warning"* .

Le manuel de la version 1.3 se présente sous la forme d'un classeur et deux cahiers séparés, un pour le manuel d'utilisation, un autre pour la mise au point de programmes (*debugger*).

Pour conclure ce préambule, rappelons que nous restons toujours ouvert à toutes les remarques que vous pourrez faire au sujet du logiciel ou de cette documentation, et que nous tenterons de répondre le mieux possible à vos suggestions ou critiques éventuelles.

_Prolog_IA

Parc Technologique de Luminy - Case 919
13288 Marseille Cedex 09 - FRANCE

Tél. : + (33) 91 26 86 36
Fax : + (33) 91 41 96 37
e-mail (net) : prolia@dcftlx.das.net
e-mail (net) : prolia@tlxf.geomail.org
Telex : Prolog 403 527 box : PrologIA

[1] Et seront retirées dès la version 1.4

Association Prolog
HERITAGE

Table des matières

Manuel de référence

Préambule ..... **i**

Différences entre la versionn 1.2 et la version 1.3 ..... **iii**

Introduction **1**

Débuter avec Prolog III **5**

1. Démarrer une session Prolog III 6
2. Utilisation d'un programme d'exemple 8
   - Chargement du programme 8
   - Suppression, édition, sauvegarde 10
3. Quelques petits exemples 12
   - Les tuples 12
   - L'arithmétique 14
   - Les booléens 16

Concepts de base **19**

1. Arbres 20
2. Constantes 23
   - Les identificateurs 24
   - Les caractères 25
   - Les valeurs booléennes 25
   - Les nombres entiers non négatifs 26
   - Les nombres flottants non négatifs 26
   - Le tuple vide 27
   - Les chaînes de caractères 27
3. Opérations 28
   - Opérations booléennes 29
   - Opérations arithmétiques 30
   - Opérations de construction d'arbres 31
   - Les listes classiques 33
4. Variables 34
5. Termes 35
   - Syntaxe 35
   - Limitations 37

© PrologIA

Exemples ... 38
Identificateurs ... 38
Identificateurs incorrects ... 39
Entiers ... 39
Entiers incorrects ... 39
Flottants ... 40
Flottants incorrects ... 40
Caractères ... 40
Chaînes de caractères ... 41
Chaînes de caractères incorrectes ... 41
Expressions numériques ... 42
Expressions numériques incorrectes ... 43
Expressions booléennes ... 43
Expressions "mixtes" arithmético-booléennes ... 44
Autres termes ... 45
Termes incorrects ... 45
Que représentent les termes ? ... 46
Affectation ... 47
Des termes particulièrement subtils… ... 48
6. Relations ... 49
Egalité et inégalité ... 50
Implication ... 52
Comparaisons numériques ... 52
Relations unaires ... 53
7. Contraintes ... 54
Syntaxe ... 54
Exemples ... 56
8. Résolution des systèmes de contraintes ... 57
9. Règles et requêtes ... 60
Syntaxe ... 60
Signification d'un programme Prolog III ... 62
L'ensemble des faits définis par un programme Prolog III ... 62
Exécution d'un programme ... 64

Arbres, tuples, chaînes et listes ... 69
1. Introduction ... 70
2. Les arbres ... 70
Opérations sur les arbres ... 71
Contraintes sur les arbres ... 74
Exemples ... 74
Restrictions sur la contrainte de taille ... 76
3. Les tuples ... 76
Opérations ... 76
Relations ... 77
Restrictions concernant les tuples ... 77
Exemples de contraintes sur les tuples ... 78
Récapitulatif des opérations et relations ... 79
4. Règles prédéfinies sur les tuples ... 80
Tailles et concaténations ... 80

Divers....81
Un exemple de programme sur les tuples....81
5. Les listes....83
Primitives sur les listes....84
6. Les chaînes....85
Primitives sur les chaînes....85
7. Exemples....87
Calcul des feuilles d'un arbre....87
Quick sort....87
Une suite périodique....89
Crible d'Eratosthène....90

Les contraintes numériques....93
1. Introduction....94
2. Généralités....94
Les nombres....94
Expressions numériques....97
Les constantes numériques....97
Les opérateurs numériques....98
Priorités des opérateurs numériques....98
Les relations....100
Relations unaires....100
Relations binaires....101
Contraintes numériques....101
Exemples de contraintes numériques....101
Restrictions....102
Forme normale....102
Les équations....103
Les inéquations....104
3. Règles prédéfinies et procédures externes spécifiques....104
Vérifications de types....106
Divers....106
4. Retardement des contraintes non-linéaires....106
5. Formats d'entrée-sortie....108
6. Exemples de programmes....108
Calcul bancaire....108
Crypto-arihmétique....109
Remplissage d'un rectangle par des carrés....110

Les contraintes booléennes....117
1. Introduction....118
2. Quelques définitions et remarques....118
Expressions booléennes....119
Priorités des opérateurs booléens....119
Contraintes booléennes....120
Remarques importantes....121
Ensembles de contraintes booléennes....122
Exemples de contraintes booléennes....122
Forme normale conjonctive....123
Affectations booléennes....125

Solution d'un ensemble de contraintes booléennes....126
Simplification de systèmes de contraintes....127
Règles prédéfinies spécifiques....128
3. Premiers exemples de programmes....129
Un ou sur deux variables....129
Un ou sur une liste....130
Au plus un de vrai....131
K éléments vrais dans une liste de n booléens....133
4. D'autres exemples....135
Pannes dans un additionneur binaire....135
Un puzzle logique....143

Retardements....147
1. Introduction....148
2. Termes connus....148
Les prédicats évaluables known, bound et free....148
3. Retardement de l'exécution d'un but....149
4. Contraintes retardées....151
Tailles retardées....151
Concaténation retardée....154
Contraintes numériques retardées....156

Le contrôle et l'environnement des programmes....161
1. Le contrôle....162
2. Expressions, variables statiques, tableaux....168
Expressions arithmétiques....168
Affectation. Tableaux.....173
3. Structuration, saisie et modification des règles....176
Familles d'identificateurs....177
Partie fermée d'une famille....178
Contexte de lecture et d'écriture....179
Modules....184
Ajout, suppression et recherche de règles....186
Manipulation de modules objets....191
4. Entrées / sorties....192
Entrées....193
Sorties....195
5. Autres éléments de l'environnement....198
Sortir de Prolog III....198
Temps, autres mesures....198

Règles prédéfinies et procédures externes....199
1. Introduction....200
Index des règles prédéfinies, procédures externes et fonctions évaluables....371

Règles prédéfinies et procédures externes (Edimbourg)....375
Identificateur - Arité....376
Règles sous forme de termes....376
Index des règles prédéfinies, procédures externes et fonctions évaluables pour la syntaxe Edimbourg....421

© PrologIA

Les syntaxes de Prolog III....423
1. Introduction....424
Choix du mode syntaxique....424
Points communs, différences....424
Conventions grammaticales....426
Jeu de caractères....427
Constantes....428
2. Syntaxe de base....432
Variables....432
Identificateurs....433
Termes....435
Expressions numériques....436
Expressions booléennes....437
Arbres et Termes....437
Contraintes....438
Règles et requêtes....440
Quelques remarques....441
Caractère espace....441
Continuation de ligne.....442
Les différents commentaires....442
3. Syntaxe d'Edimbourg....443
Variables....443
Identificateurs....444
Termes....445
Opérateurs....445
Les termes....447
Expressions numériques....451
Expressions booléennes....452
Arbres et Termes....452
Contraintes....453
Règles et requêtes....456
Quelques remarques....458
Caractère espace....458
Continuation de ligne.....458
Les différents commentaires....459
Opérateurs Edimbourg....459
Etrangetés....459
4. Remarques générales....460
Différences des modes....460
Cohabitation des deux syntaxes....461

Primitives Graphiques....463
1. Introduction et conventions....464
2. Primitives de gestion des fenêtres....466
Rafraîchissement de la fenêtre graphique....473
3. Dessin et positionnement....473

4. Usage de la souris 478
Règles pour manipuler les fenêtres à l'aide du clavier et de la souris 478
Repérage de la souris 479
5. Modes de dessin et d'écriture 481
6. Primitives spéciales de lecture et d'écriture 485
Simulation de boutons 485
Saisie de texte 486
Boîtes de choix 489
Impression de fenêtres 492
7. Primitive de description de menus 493
8. Gestion des boîtes de dialogue 498
Définition d'un dialogue standard 498
Extensions utilisateur 510
9. Boutons actifs 515
10. Primitives de manipulation de fichiers 516
Index des primitives graphiques 519

Annexes 523

A. Liste des messages d'erreur de Prolog III 523

B. Liste des règles prédéfinies par catégorie 529
Règles prédéfinies Prolog III 529
Arbres, listes, chaînes et tuples 529
Contrôle 529
Contraintes numériques 530
Conversion 530
Environnement 530
Entrées/Sorties 531
Gestion des règles et des identificateurs 532
Vérification de types 532
Règles prédéfinies Edimbourg 533
Arbres, listes, chaînes et tuples 533
Contrôle 533
Evaluation de fonctions prédéfinies 533
Entrées/Sorties 534
Gestion des règles et des identificateurs 534
Vérification de types 534

C. Quelques programmes Prolog III en syntaxe Edimbourg 535
1. Un petit exemple : menu.p3e 536
Le programme : 536
Une Question : 536
2. Calculs sur les grands nombres : fact.p3e 537
Le programme : 537
Questions : 537
3. Calculs bancaires : bank.p3e 538
Le programme : 538
Questions posées avec des nombres entiers ou flottants : 538
4. Manipulation d'arbres : leaves.p3e 539

© PrologIA

Le programme :....539
Questions :....539
5. Un peu de logique : god.p3e....540
Le programme :....540
LA question :....540
6. Un problème de Lewis Caroll : lewis.p3e....541
Le programme :....541
Exemples de questions :....542
7. Détection de panne dans un additionneur : circuit.p3e....543
Le programme :....543
Premier jeu de questions :....543
Second jeu de questions :....544
8. Un puzzle numérique : send.p3e....545
Le programme :....545
La question :....545
9. Un problème de remplissage : rectangle.p3e....546
Le programme :....546
Une question :....546

Index du manuel de référence....547

Manuel d'utilisation

Utilisation de Prolog III....1001
1. Installation de Prolog III....1002
Matériel et logiciel requis....1002
Contenu du kit d'installation....1002
Fichiers indispensables pour utiliser Prolog III....1003
Fichiers annexes....1004
Procédure d'installation....1005
Sur disque dur....1005
Sur Disquettes....1005
2. Utilisation de Prolog III....1005
Activation de Prolog III....1005
Paramétrisation de l'interprète Prolog III....1006
Interruption d'un programme....1007
3. Spécificités de Prolog III Macintosh....1008
Valeurs extrêmes des constantes arithmétiques....1008
Espaces et tailles par défaut....1009
4. A propos de l'environnement....1009
Utilisation d'un éditeur hôte....1009
5. Ajout de règles prédéfinies....1010
Marche à suivre....1011
Appel d'une règle externe....1012
Procédures de transfert de données....1013
Premier protocole de transfert de données....1014
Fonctions générales :....1014

Fonctions de conversion :.... 1016
Second protocole de transfert de données....1018
Transfert de données simples de Prolog vers C.... 1018
Transfert de données simples de C vers Prolog.... 1020
Un exemple complet....1022
6. Appeler Prolog III depuis le langage C....1025
Introduction....1025
Description des fonctions....1025
Initialisation....1025
Appel....1026
Terminaison....1027
Un petit exemple....1028
Listings....1030
Index Général....1033

Manuel de mise au point

Débuggeur et mise au point de programmes Prolog III....2001
1. Introduction....2002
2. Trace d'exécution....2003
3. Aperçu du modèle des boites....2004
Quelques définitions....2006
4. Les commandes du débuggeur....2007
1. Le langage de commande....2007
2. Points d'impression et points d'arrêt....2008
3. Classifications des commandes....2008
4. Progression....2009
5. Informations....2011
6. Installer des points d'arrêt....2015
7. Modifier le déroulement de l'exécution....2018
8. Gestion des options du débuggeur....2022
9. Commandes annexes....2028
5. Règles prédéfinies de mise au point....2031
6. Mise au point avancée....2031
1. Reprendre une démonstration (retry)....2031
2. Récupérer une erreur....2033

Association Prolog
HERITAGE

Prolog III

Version 1.3

Manuel de Référence

Décembre 1991

© PrologIA

Association Prolog
HERITAGE

Garantie et responsabilités

PrologIA n'offre aucune garantie, expresse ou tacite, concernant ce manuel ou le logiciel qui y est décrit, ses qualités, ses performances ou sa capacité à satisfaire à quelque application que ce soit.

PrologIA ne pourra être tenue responsable des préjudices directs ou indirects, de quelque nature que ce soit, résultant d'une imperfection dans le programme ou le manuel, même si elle a été avisée de la possibilité que de tels préjudices se produisent. En particulier, elle ne pourra encourir aucune responsabilité du fait des données mémorisées ou exploitées, y compris pour les coûts de récupération ou de reproduction de ces données.

L'acheteur a toutefois droit à la garantie légale dans les cas et dans la mesure seulement où la garantie légale est applicable nonobstant toute exclusion ou limitation.

Droits d'auteur

Ce manuel et le logiciel qu'il décrit sont protégés par les droits d'auteur. Au terme de la législation traitant de ces droits, ce manuel et ce logiciel ne peuvent être copiés ou adaptés, en tout ou en partie, sans le consentement écrit de PrologIA, sauf dans le cadre d'une utilisation normale ou pour faire une copie de sauvegarde. Ces exceptions n'autorisent cependant pas la confection de copies à l'intention d'un tiers, que ce soit ou non pour les vendre.

Prolog III est une marque déposée de PrologIA.

Association Prolog
HERITAGE

Introduction

Ce manuel entend répondre à un certain nombre des besoins légitimes de l'utilisateur d'un nouveau langage de programmation. Ces besoins sont d'autant plus nombreux que certains des concepts utilisés couramment dans les programmes Prolog III ne sont présents dans aucun autre langage. Ainsi, nous présenterons ici, outre les caractéristiques purement syntaxiques qui décrivent les programmes, un certain nombre de considérations, parfois assez complexes, qui permettent, en cernant les mécanismes sous-jacents qui régissent l'unification, de programmer mieux, plus clairement, plus efficacement, et peut être de programmer tout court. Dans cet ordre d'idées, de nombreux exemples seront fournis et développés, au fur et à mesure des besoins rencontrés.

Prolog III, tout comme son prédécesseur Prolog II, a ceci de remarquable qu'il possède un modèle théorique complet qui justifie, entre autres, que tout programme respectant sa syntaxe est exécuté d'une manière unique et prévisible. Le lecteur intéressé par les aspects théoriques de la question pourra se référer à de nombreux articles et ouvrages publiés sur le sujet. Toutefois, pour des raisons de clarté, nous avons préféré ici nous placer dans une optique relativement informelle, qui devrait également s'avérer plus accessible. Nous espérons que les adeptes des présentations théoriques ne nous en tiendront pas rigueur.

Enfin, le lecteur peu habitué à Prolog pourra se sentir quelque peu malmené par ce manuel, dont le but principal est de présenter Prolog III et la programmation logique par contraintes, au détriment quelquefois d'une approche didactique et circonstanciée des notions couramment utilisées en Prolog. Là encore, un certain nombre d'ouvrages, dont “Prolog” chez Inter Editions, vous permettront de cerner les notions de base de la programmation logique.

© PrologIA

Quelques mots sur l'organisation de ce manuel. Il est divisé en trois parties : le manuel de référence, les manuels d'utilisation et de mise au point de programmes, ces derniers sous forme de cahiers séparés.

Le manuel de référence.

Notre intention, lors de la rédaction de ce manuel, a été de d'offrir au lecteur une approche à plusieurs niveaux; ces niveaux pouvant différer, selon chaque lecteur, aussi bien par l'état de ses connaissances sur Prolog III que par le contexte d'utilisation à un instant donné.

Une première prise de contact pose les grands principes de Prolog III au travers de l'utilisation de petits exemples. Cette prise de contact pourra s'effectuer devant le clavier de votre machine favorite.

Un certain nombre de chapitres forment ensuite ce que l'on pourrait, avec un soupçon de prétention, appeler un "cours" sur Prolog III. On y détaille les concepts à la base du langage, pour revenir dans le détail sur les trois principaux aspects : les tuples, le numérique et les booléens. Le dernier de ces chapitres explicite la notion de retardement. Des exemples émaillent cette présentation générale pour en faciliter la compréhension.

Le manuel se poursuit par une description de tout ce qui est utile pour le programmeur, le contrôle, l'environnement, les entrées/sorties.

Ensuite on trouvera une liste, que nous espérons complète et que nous avons voulu alphabétique, des primitives de Prolog III; tout d'abord la liste des primitives communes aux deux syntaxes, puis celles propre à la syntaxe de type Edimbourg. Ces primitives sont présentées à raison d'une par page, et nous avons tenté de multiplier, à partir de chaque entrée, les accès possibles à d'autres primitives, ou même à d'autres parties du manuel.

Le chapitre "Syntaxes" présente ensuite les syntaxes de Prolog III. Deux syntaxes sont en effet admises, une syntaxe originale proche de Prolog II, et une syntaxe anglaise de type Edimbourg.

Enfin le dernier chapitre, "Primitives graphiques", donne une description détaillée des primitives d'environnement qui permettent de gérer aussi bien

© PrologIA

du graphique, des fenêtres, ou des menus déroulant, que des interruptions souris, ou des objets structurés.

Avant de clore le manuel de référence par un certain nombre d'index, généraux ou spécifiques, on trouvera trois annexes, une première concernant la liste des messages d'erreur de Prolog III, une seconde contenant la liste des règles prédéfinies et enfin une troisième reprenant les exemples du manuel, cette fois ci en syntaxe Edimbourg.

Le manuel d'utilisation.

La partie "manuel d'utilisation" présente les différentes spécificités de Prolog III sur la machine que vous utilisez. On y trouvera également les informations relatives à l'installation de Prolog III sur votre machine ainsi qu'à l'ajout de règles prédéfinies ou encore aux appels externes.

A la fin de ce cahier, on donne un récapitulatif des index communs aux manuels de référence et d'utilisation.

Le manuel de mise au point de programmes Prolog III.

Cette partie décrit le debugger Prolog III à travers son langage de commande et ses primitives. Une présentation de son modèle est donnée en début de chapitre, et une approche progressive par des exemples démonstratifs amène naturellement à une description plus détaillée de son fonctionnement et de primitives plus complexes.

ststs

© PrologIA

Association Prolog
HERITAGE

Débuter avec Prolog III

1. **Démarrer une session Prolog III**
2. **Utilisation d'un programme d'exemple**
   - **Chargement du programme**
   - **Suppression, édition, sauvegarde**
3. **Quelques petits exemples**
   - **Les tuples et les chaînes de caractères**
   - **L'arithmétique et les contraintes numériques**
   - **Les booléens et les formules**

Que trouve-t-on dans ce chapitre ?

Ce chapitre vous propose un premier contact avec Prolog III, à travers un petit nombre d'exercices dont la principale caractéristique est d'être très simples. Nous ne cherchons pas à introduire les concepts fondamentaux de Prolog III, ce sera l'objet des chapitres suivants, mais uniquement la pratique simple du système Prolog III : l'activer, composer un programme, l'essayer, le modifier, etc…

Décembre 1990

© PrologIA

1. Démarrer une session Prolog III

La manière d'entrer dans l'univers de Prolog III dépend du système d'exploitation de votre ordinateur. Sur UNIX, VMS, MS-DOS, etc… il vous suffit de composer la commande suivante (on suppose que **$** est le « prompt » de votre système) :

```
$  prolog3
```

Prolog imprime :

```
PROLOG III, v1.2 Decembre (C) PrologIA 1989,90

>
```

Sur Macintosh, vous devez retrouver le fichier contenant le logiciel Prolog III et l'ouvrir. Pour cela, “cliquez” deux fois sur l'icône ci-contre. L'activation de Prolog III demande un court instant. Une fenêtre nommée *console* s'ouvre, dans laquelle est affiché un texte de la forme :

```
PROLOG III, v1.2 Decembre (C) PrologIA 1989,90

>
```

A partir d'ici, le comportement de Prolog III est pratiquement le même sur toutes les machines, tout au moins pour ce qui concerne les exercices qui vont nous intéresser dans ce chapitre.

Le caractère **>** sert de « prompt » à Prolog III et indique à l'utilisateur qu'il est en attente d'une commande. Une commande peut être aussi bien une règle qu'une requête. Donnons une requête simple pour commencer :

© PrologIA

```
> outml("Bonjour  !");
Bonjour !
{}
>
```

Prolog imprime des accolades pour exprimer que la requête s'est exécutée avec succès. Entrons maintenant un premier petit programme : les trois règles qui expriment que Jacques, Annie et Rémi habitent respectivement à Paris, Marseille et Paris :

```
> habite_a(Jacques,Paris)  ->;
> habite_a(Annie,Marseille)  ->;
> habite_a(Remi,Paris)  ->;
>
```

Les règles sont insérées dans l'ordre où on les a tapées. Entrons la commande *list* (qui affiche toutes les règles données par l'utilisateur).

```
> list;
habite_a(Jacques, Paris) ->;
habite_a(Annie, Marseille) ->;
habite_a(Remi, Paris) ->;
{}
>
```

Faisons quelques essais avec ce programme : où habite Jacques ?

```
> habite_a(Jacques,x);
{x=Paris}
>
```

Qui habite à Paris ?

```
> habite_a(x,Paris);
{x=Jacques}
{x=Remi}
>
```

Qui habite où ?

```
> habite_a(x,y);
{x=Jacques, y=Paris}
{x=Annie, y=Marseille}
{x=Remi, y=Paris}
>
```

A chaque fois, la réponse du système est l'ensemble des valeurs à donner aux variables figurant dans la question pour que la relation correspondante soit satisfaite. Pour terminer la session, on tape la commande :

```
> quit;
Prolog Quit ...
$
```

et on se retrouve sous l'interprète de commandes du système d'exploitation.

2. Utilisation d'un programme d'exemple

Chargement du programme

Dans ce qui suit, on utilisera le programme contenu dans le fichier *menu.p3*. Il s'agit du calcul des repas légers, réécrit en Prolog III, très connu mais très pédagogique. Ce programme est livré avec l'interprète. Il suffit de lancer Prolog III comme précédemment, puis de charger le fichier *menu.p3* en tapant :

```
> input("menu.p3");
{}
>>>>>>>>>>>> Erreur 17: Fin de fichier inattendue
>
```


© PrologIA

Le but *input("menu.p3")* impose que l'entrée courante bascule sur le fichier *menu.p3*. Les commandes y sont lues tour à tour, exécutés s'il s'agit de requêtes, codées en mémoire si ce sont des règles. Lorsque le fichier d'entrée est épuisé, le message : «Erreur 17: Fin de fichier inattendue» apparaît et l'entrée courante bascule sur l'entrée précédente qui, en l'occurrence, est le clavier. Le message d'erreur n'aurait pas été affiché si à la fin du fichier en question on avait trouvé le but *close_input*. On peut visualiser les règles insérées par :

```
> list;
RepasLeger(h,p,d) ->
     HorsDoeuvre(h,i)
     Plat(p,j)
     Dessert(d,k),
     {-k -i -i +10 >= 0 ,
      i >= 0 ,
      j >= 0 ,
      k >= 0 };

HorsDoeuvre(radis,1) ->;
HorsDoeuvre(pate,6) ->;

Plat(p,i) ->
     Viande(p,i);
Plat(p,i) ->
     Poisson(p,i);

Dessert(fruit,2) -> ;
Dessert(glace,6) -> ;

Viande(boeuf,5) -> ;
Viande(porc,7) -> ;

Poisson(sole,2) -> ;
Poisson(thon,4) -> ;

{}
>
```

Remarquons tout d'abord que ces règles ne sont pas tout à fait identiques à celles qui se trouvent dans le fichier *menu.p3*. Peu importe, elles leur sont équivalentes. Prolog a codé les règles prises dans le fichier *menu.p3* sous une forme canonique et ne peut plus les décoder pour produire le texte initial car

il existe une infinité de textes initiaux possibles. Le fichier *menu.p3* n'est pas modifié par Prolog.

Faisons un essai avec ce programme. Demandons de quoi est constitué un repas léger; cela se traduit par :

```
> RepasLeger(h,p,d);
{ h = radis, p = boeuf, d = fruit}
{ h = radis, p = porc, d = fruit}
{ h = radis, p = sole, d = fruit}
{ h = radis, p = sole, d = glace}
{ h = radis, p = thon, d = fruit}
{ h = pate, p = sole, d = fruit}
>
```

Suppression, édition, sauvegarde

Si nous voulons changer notre menu en remplaçant les règles :

```
Viande(boeuf,5) -> ;
Viande(porc,7) -> ;
```

par les règles :

```
Viande(veau,4) -> ;
Viande(boeuf,5) -> ;
Viande(porc,7) -> ;
```

il nous suffit de supprimer le paquet de règles ayant pour nom *Viande,* puis d'insérer le nouveau paquet :

```
> suppress(Viande);
{}
> Viande(veau,4) -> ;
> Viande(boeuf,5) -> ;
> Viande(porc,7) -> ;
>
```


© PrologIA

observons le paquet de nom *Viande* en utilisant la commande *list*:

```
> list(Viande);
Viande(veau,4)
Viande(boeuf,5)
Viande(porc,7)
{}
>
```

Exécutons à nouveau le programme :

```
> RepasLeger(h,p,d);
{ d = radis, p = veau, h = fruit}
{ d = radis, p = boeuf, h = fruit}
{ d = radis, p = porc, h = fruit}
{ d = radis, p = sole, h = fruit}
{ d = radis, p = sole, h = glace}
{ d = radis, p = thon, h = fruit}
{ d = pate, p = sole, h = fruit}
>
```

Si nous voulons sauvegarder nos modifications dans un fichier, par exemple *menubis.p3*, il nous faut taper les commandes suivantes :

```
> output("menubis.p3") list close_output;
{}
>
```

La commande *output* fait basculer la sortie courante sur le fichier donné en argument, *list* va nous lister notre programme sur cette sortie et *close_output* ferme la sortie courante et bascule sur la sortie précédente, dans notre cas, l'écran. Nous pouvons ensuite quitter la session, toujours par :

```
> quit;
$
```

Une fois revenu sous notre système d'exploitation, on peut vérifier que le fichier *menubis.p3* a bien été créé, comprenant les modifications que nous avons faites plus haut.

3. Quelques petits exemples

Prolog III est un langage qui manipule, outre les arbres que l'on supposera bien connus, les tuples, les expressions numériques linéaires, les formules booléennes. Tous ces objets peuvent être solutions de système de contraintes.

Nous allons voir dans ce paragraphe, à travers quelques petits exemples comment manipuler ces différents objets.

ATTENTION. Quand on "joue" avec Prolog III, comme vous allez probablement le faire maintenant, on tape un petit programme, qu'on essaye, puis un autre, et encore un autre, etc… Or Prolog III conserve tous ces programmes en mémoire, et on finit immanquablement par obtenir des erreurs qui traduisent des conflits de noms comme « Erreur 46: Regle deja definie ». Pour l'éviter remettez à zéro l'espace des règles, entre deux exemples, en exécutant la commande :

```
> kill_module("");
{}
>
```

Les tuples et les chaînes de caractères

Un tuple est une suite d'objets Prolog. Le tuple noté $<a_1,a_2,a_3>$ représente la suite de longueur 3 formée des éléments a_1, a_2, a_3, quelle que soit la nature de ces éléments. Le tuple vide s'écrit naturellement <>. On peut indiquer qu'une variable x est de type tuple en ajoutant la contrainte *x !tuple*. On peut aussi imposer que x soit un tuple de longueur n donnée par $x :: n$. L'opération de base sur les tuples est la concaténation, notée par . (point). Observons les exemples suivants :

```
> {x = <1,2,3>.<4,5>};
{x=<1,2,3,4,5>}
> {x = y.z, y::2, y = <1,2>};
{x=<1,2>.z, y=<1,2>}
> {x = y.z, y::2};
```


© PrologIA

```
{x=y.z, y::2}
>
```

N.B. Les exemples précédents sont donnés à titre explicatif et la réponse fournie par la machine peut avoir une forme différente, mais équivalente.

Si on souhaite obtenir le premier élément *x* d'un tuple *l*, il suffit de l'unifier avec un terme de la forme *<x>.r* , qui dénote la concaténation d'un tuple *<x>* à un seul élément et d'un tuple *r* quelconque. Ceci nous permet d'écrire les deux règles *element_de* en Prolog III :

```
> element_de(x,<x>.r) ->;
> element_de(x,<y>.r) -> element_de(x,r) , {x#y};
>
```

puis on peut demander :

```
> element_de(1,  <3,5,1>);
{}
> element_de(x,  <3,5,1>);
{x=3}
{x=5}
{x=1}
> element_de(9,  <3,5,1>);
>
```

Les chaînes de caractères sont des tuples dont les éléments sont des caractères. La chaîne notée *"abc"* est le tuple de caractères *<`a`,`b`,`c`>* .

On peut mélanger les chaînes et les tuples, notamment à l'aide de l'opérateur de concaténation. On peut imposer à une variable de type chaîne une longueur *n* connue (le nombre de caractères) en installant la contrainte *x :: n* .

```
> {x=y.<`-`>."au-".<`f`,`e`,`u`>,  y="pot"};
{x="pot-au-feu", y="pot"}
>
```

L'arithmétique et les contraintes numériques

Voyons le problème suivant : on dispose d'un certain nombre de chats, d'oiseaux, les nombres de têtes et de pattes et on demande d'exprimer les relations liant ces nombres. Si on appelle *c* le nombre de chats, *o* le nombre d'oiseaux, *t* le nombre de têtes, *p* le nombre de pattes, les relations sont les suivantes :

$$t = c + o \text{ et } p = 4c + 2o$$

et s'expriment naturellement en Prolog III :

```
> {t=c+o,  p=4c+2o};
{t=o+c, p=2o+4c}
>
```

Prolog répond par un système d'équations, ce qui signifie que le système est soluble. Allons un peu plus loin en donnant des valeurs à certaines variables. Comme il s'agit d'un système de deux équations à quatre inconnues, nous allons fixer les valeurs de deux des variables pour obtenir les valeurs des deux autres. Fixons le nombre de pattes à 14 et le nombre de têtes à 5 :

```
> {t=c+o,  p=4c+2o,  p=14,  t=5};
{t=5,c=2,o=3,p=14}
>
```

Prolog nous dit cette fois-ci que la solution est unique en nous fournissant les valeurs des variables pour lesquelles l'ensemble de contraintes est vérifié. On aurait pu bien évidemment fixer n'importe quel couple de variables pour avoir une solution unique. Avant de continuer avec cet exemple nous allons, afin de ne plus donner explicitement le système d'équations, déclarer dans Prolog une règle liant les quatre variables et les équations. Par la suite, nous n'utiliserons plus que cette relation. Entrons:

```
> ChatsOiseaux(c,o,p,t)  -> , {t=c+o, p=4c+2o};
>
```


© PrologIA

Nous venons d'introduire dans Prolog une règle dont la sémantique est : *sous réserve que le système de contraintes soit vérifié, la relation ChatsOiseaux(c, o, p, t) est vraie*. On peut à présent donner les requêtes suivantes :

```
> ChatsOiseaux(c,o,14,5);
{c=2, o=3}
> ChatsOiseaux(1,o,p,5);
{o=4, p=12}
> ChatsOiseaux(1,1,2,4);
> ChatsOiseaux(c,o,6,4);
{c=-1, o=5}
>
```

Prolog ne répond rien à la troisième requête car elle n'est pas soluble. A la quatrième, une des variables de la solution est négative. En effet, lors de la formalisation de notre problème, nous n'avons à aucun moment précisé que les solutions devaient être positives ! Nous allons donc réécrire la relation ChatsOiseaux en imposant aux variables figurant dans le système d'équations de ne pas être négatives. Auparavant, nous enlevons la relation ChatsOiseaux en donnant la requête :

```
> suppress(ChatsOiseaux);
{}
>
```

puis nous écrivons la nouvelle relation :

```
> ChatsOiseaux(c,o,p,t)  ->,
    {t=c+o,  p=4c+2o,  c>=0,  o>=0,  p>=0,  t>=0};
>
```

et nous demandons :

```
> ChatsOiseaux(c,o,6,4);
>
```

Prolog nous dit qu'il n'y a pas de solution à cette requête, ce que nous voulions. Mais que se passe-t-il si nous fournissons un nombre de pattes et de têtes tels qu'il n'existe pas de solution entière ?

```
> ChatsOiseaux(c,o,7,3);
```

```
{c=1/2, o=5/2}
>
```

nous nous retrouvons avec $\frac{1}{2}$ chat et $\frac{5}{2}$ d'oiseaux.... Nous n'avions en effet jamais précisé que nous voulions des solutions entières. Prolog résout les équations arithmétiques dans l'ensemble **Q** des nombres rationnels. Nous arrêterons là cet exemple car il n'existe pas de contraintes imposant à une variable d'être entière; il nous faudra utiliser un prédicat prédéfini pour réaliser cette vérification.

Les booléens et les formules

Prolog manipule aussi les formules et les systèmes de contraintes booléennes. Il en vérifie à tout instant la satisfaisabilité et donne un système « simplifié » comme solution. Nous ne nous étendrons pas ici sur la notion de simplifié ; disons juste que le système solution est équivalent à celui fourni en entrée. Donnons par exemple ces trois relations sur les trois variables *a*, *b*, *c* :

```
> {a=>b,  b=>c,  c=>a};
{b=>c, c=>a, a=>b}
>
```

Rappelons que ces trois relations *implique* indiquent que les trois variables *a*, *b*, *c* sont équivalentes et ont donc même valeur de vérité, sans pour autant que celle-ci soit connue. Prolog répond par un système identique à celui que nous avons fourni au départ, ce système étant en fait déjà "simplifié".

Imposons maintenant que l'une des trois variables *a* ou *b* ou *c* soit vraie. Pour cela on ajoute au système précédent l'équation *a* | *b* | *c* = 1' qui dit que la formule «*a* ou *b* ou *c* » est vraie (1' est la constante Prolog *vrai*, et 0' la constante *faux*).

```
> {a=>b,  b=>c,  c=>a,  a|b|c  =  1'};
{a=1',b=1',c=1'}
>
```


© PrologIA

Cette fois-ci, la solution est unique et Prolog nous répond que toutes les variables prennent la valeur "vrai".

ststs

Association Prolog
HERITAGE

Concepts de base

1. **Arbres**
2. **Constantes**
3. **Opérations**
4. **Variables**
5. **Termes**
6. **Relations**
7. **Contraintes**
8. **Résolution des**
9. **Règles et requêtes**

Que trouve-t-on dans ce chapitre ?

Quels sont les concepts fondamentaux de Prolog III ? Quels objets le langage manipule-t-il ? Quelle est la nature des contraintes qui pèseront sur ces objets ? Qu'est-ce qu'un programme Prolog III signifie pour le programmeur ? Comment la machine l'exécute-t-elle ? Selon quelle syntaxe ces objets et contraintes doivent-ils être exprimés ?

Il y a une différence de nature, et non de degré, entre la plupart des autres langages de programmation et Prolog III. On ne peut aborder ce dernier sans examiner auparavant un certain nombre de concepts originaux à la base du langage, indispensables pour une utilisation optimale, parfois même une utilisation tout-court, de celui-ci.

Ce chapitre présente l'ensemble des spécifications de Prolog III, d'une manière assez informelle, accessible quel que soit le degré de familiarité que vous entretenez déjà avec la programmation logique.

Nous y définissons les objets manipulés par le langage, puis les opérations et les relations qui permettent d'établir des contraintes sur ces objets. Ensuite, nous présentons ce qu'est un programme Prolog III, ainsi que la manière dont il est exécuté.

© PrologIA

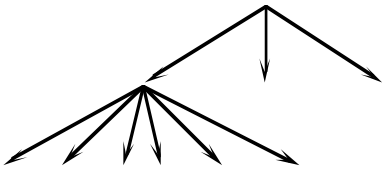
1. Arbres

« Quelle est la nature des objets manipulés ? » C'est une des premières questions auxquelles il nous faut répondre pour expliquer Prolog III. Le langage est tout-à-fait cohérent et uniforme :

TOUS LES OBJETS MANIPULéS PAR PROLOG III SONT DES ARBRES

Certains sont très complexes, voire infinis, tandis que d'autres sont réduits à un unique nœud. Certains ont une dénomination particulière, alors que d'autres ne peuvent être designés qu'au moyen d'une opération qui les construit ou d'un système de contraintes dont ils sont solution.

N'ayant pas à reproduire les démonstrations formelles qui justifient les algorithmes du cœur de Prolog III, nous ne donnerons pas ici de définition rigoureuse des arbres. La notion intuitive habituelle, qui assimile le concept d'arbre à celui d'*ensemble organisé hiérarchiquement* nous suffira amplement.



Cette hiérarchie est incarnée dans un ensemble de positions, ou *nœuds*, et un ensemble de *flèches* reliant ces positions. On dit que les nœuds auxquels aboutit une flèche issue d'un nœud *n* sont les *fils* de *n*. L'ensemble des fils d'un nœud donné est toujours ordonné, ne serait-ce que par l'ordre dans lequel ces nœuds sont pratiquement représentés. Le nœud qui est au sommet de la hiérarchie s'appelle la *racine*, ou *nœud initial* de l'arbre.

© PrologIA

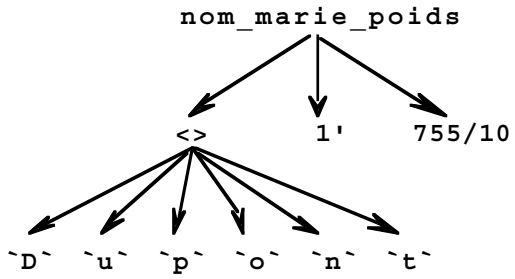
Chaque nœud d'un arbre est à son tour racine d'un arbre, appelé *sous-arbre* de l'arbre initial, défini par ce nœud, ses fils, les fils de ses fils, etc… ainsi que les flèches qui émanent de tous ces nœuds.

UN ARBRE COMPORTE UNE ÉTIQUETTE ET UNE SUITE FINIE DE SOUS-ARBRES.

A chaque nœud d'un arbre Prolog III est associée une étiquette. Les étiquettes possibles sont :

- les identificateurs
- les caractères
- les valeurs booléennes `0'` et `1'`
- les nombres
- le double signe `<>`

Voici un arbre de Prolog III :



Le nombre des fils d'un nœud est toujours fini et indépendant de la nature de son étiquette. Bien entendu, la sémantique de cette dernière, ce qu'elle signifie pour le programmeur, peut faire qu'un nœud n'ait de sens qu'assorti d'un nombre déterminé de fils, mais cela n'est pas connu des mécanismes de base de Prolog III.

Le nombre des fils d'un nœud peut être nul : un arbre réduit à un seul nœud est appelé une *feuille*. Prolog III ne fait pas de distinction entre une feuille et l'étiquette qu'elle porte. Par conséquent, les identificateurs, les caractères, les valeurs booléennes et les nombres sont considérés comme des cas particuliers d'arbres.

Tous ces éléments peuvent servir à étiqueter un nœud, un cas particulièrement fréquent étant celui où l'étiquette est un identificateur, comme dans l'exemple dessiné ci-dessus. Fort souvent de tels arbres représentent des relations. Pour l'arbre dont nous parlons, ce pourrait être « *s'appelle Dupont, est marié et pèse 75,5 Kg* ».

Un autre cas fréquent est celui où l'étiquette du nœud initial de l'arbre n'a aucune signification pour le programmeur, qui ne s'intéresse donc qu'à la suite des fils. Prolog III permet d'utiliser alors pour étiquette le double signe conventionnel `<>`. De tels arbres sont appelés *PIII-tuples* ou, plus simplement, *tuples*, et ils implantent donc la notion de suite finie d'arbres. Dans cet esprit, l'arbre réduit au symbole `<>` représente la suite vide. Les *tuples* sont expliqués plus en détail au chapitre « Arbres, tuples, chaînes et listes »

Grâce au puissant opérateur dont ils sont munis, qui traduit l'opération de concaténation des suites finies, les tuples constituent un outil souple et puissant pour la définition de nombreuses structures de données, alliant la généralité des listes et l'efficacité des vecteurs. Ils peuvent donc avantageusement remplacer la notion habituelle de liste, comme elle existe en Lisp ou dans d'autres Prolog.

Cependant ces listes classiques, définies à partir de la notion de paire pointée, peuvent être aussi utilisées en Prolog III, y compris à travers leur syntaxe habituelle, dite d'Edimbourg.

Dans les programmes Prolog III, les arbres sont représentés par des formules appelées *termes*. La syntaxe des termes apparaîtra progressivement tout au long des paragraphes suivants ; elle est résumée au chapitre « Les syntaxes de Prolog III ».

© PrologIA

Les arbres et les termes ne sont pas la même chose : les éléments du domaine de Prolog III sont des arbres, les entités syntaxiques qui les représentent sont des termes. Or, dans l'explication de Prolog III il ne sera guère question que de ces derniers, car ils sont la seule expression écrite des arbres en accord avec la syntaxe du langage. Ceci semble encourager une certaine confusion des deux notions. Nous ne saurions trop mettre en garde le lecteur contre une telle confusion, car elle présente de nombreux inconvénients et notamment celui d'empêcher la compréhension de notions parmi les plus importantes.

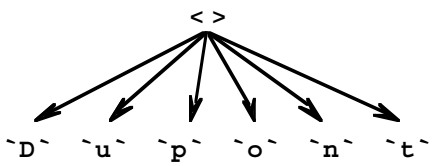
Cela étant dit, il y a un cas dans lequel on confond un arbre et le terme qui le représente : il s'agit des *constantes*, qui sont par définition les arbres représentés par des termes simples, dont l'expression ne fait pas intervenir d'opérateur, ainsi que l'explique le paragraphe suivant.

2. Constantes

Parmi les éléments du domaine de Prolog III, c'est-à-dire les arbres, les *constantes* désignent ceux qui ont reçu une dénomination particulière. Comme nous l'avons dit, nous ne ferons pas de distinction entre une constante et le terme qui la représente.

Il ne suffit pas d'être une feuille pour être une constante : un arbre réduit à un unique nombre fractionnaire, comme `755/10`, est une feuille, mais il ne possède pas de nom spécifique et il ne peut être exprimé qu'à travers la division dont il est le résultat. Bien entendu, `151/2` est une autre manière de spécifier le même nombre rationnel.

Inversement, toutes les constantes ne représentent pas nécessairement des feuilles. Ainsi une suite de caractères comme



n'est certainement pas atomique ; cependant, Prolog III la considérera comme une constante, dès lors que l'on utilisera la notation particulière des chaînes, `"Dupont"`. Bien entendu, vous pouvez représenter le même arbre par le terme `<`D`,`u`,`p`,`o`,`n`,`t`>`.

Les constantes connues sont :

Les identificateurs

Les identificateurs sont les constantes symboliques, comme

```
pierre
repas_leger
calcul12
```

Un identificateur est une suite de lettres, de chiffres et des deux caractères apostrophe (') et blanc souligné (_) qui n'a pas la syntaxe d'un nom de variable, c'est-à-dire qui commence par au moins deux lettres. Une description plus précise de la syntaxe des identificateurs est donnée au chapitre « Les syntaxes de Prolog III ».

En réalité la syntaxe des identificateurs est bien plus complexe que celle des exemples montrés ici, puisque la notation complète d'un identificateur comporte un *qualifieur (ou préfixe)* qui indique le module auquel l'identificateur appartient. L'ensemble des modules intervenant dans un programme Prolog III réalise une partition de l'espace des noms, lui permettant d'être gigantesque tout en restant facile à appréhender par le programmeur. Cependant, l'existence par ailleurs d'un *contexte de lecture-écriture* permet le plus souvent de n'utiliser que des notations simplifiées pour les identificateurs.

Les notions de *module*, de *qualifieur* et de *contexte de lecture-écriture* sont expliquées au chapitre « Le contrôle et l'environnement des programmes »

© PrologIA

Les caractères

Tous les caractères disponibles sur votre machine peuvent être utilisés dans un programme Prolog III. Les caractères imprimables peuvent être exprimés en les encadrant par des « anti-apostrophes » :

```
`A`,  `a`,  `<`
```

Les caractères non imprimables et ceux ayant un rôle spécial peuvent être indiqués au moyen du caractère d'échappement *anti-slash* (\ ), selon une convention proche de celle utilisée dans la communauté UNIX. Par exemple :

`` `\`` `` indique le caractère `
`` `\n` `` indique le caractère “nouvelle ligne”
`` `\x41` `` indique le caractère `A` (de code ASCII 41 en hexadécimal)
`` `\101` `` indique le caractère `A` (de code ASCII 101 en octal)
`` `\\` `` indique le caractère \ lui-même

La syntaxe complète des caractères est donnée au chapitre « Les syntaxes de Prolog III ».

Les valeurs booléennes

Les valeurs booléennes sont les deux éléments du domaine de l'algèbre de Boole classique :

```
0'
1'
```

Nous ne les interpréterons pas en termes de *vrai* ou de *faux*, réservant ces mots pour parler des contraintes (nous dirons qu'une contrainte est vraie ou fausse, pour indiquer qu'elle est ou n'est pas satisfaite). Les contraintes sont expliquées ci-dessous, au § 7.

Les nombres entiers non négatifs

```
0
1
2
1991
81591528324789773434561126959611589427200000000
```

On notera que les entiers négatifs ne sont pas considérés comme des constantes, puisqu'on peut les exprimer comme résultat d'une opération, le changement de signe, appliquée à un entier positif ou nul. De la même manière, les nombres fractionnaires ne sont pas des constantes non plus, ils sont représentés à l'aide des opérations de division et de changement de signe, appliquées à des entiers.

Les nombres rationnels sont représentés dans la machine en *précision parfaite*, c'est-à-dire avec autant de chiffres que leur expression exacte le requiert. A la condition, bien entendu, que cela ne dépasse pas la taille de la mémoire de votre machine.

Les nombres flottants non négatifs

Exemples :

```
12.345
0.5
314.15926535e-2
1.2E12
```

La représentation interne, et donc la précision et l'étendue des nombres flottants, sont déterminés par les particularités de votre machine. Leur syntaxe coïncide avec celle utilisée dans la plupart des langages qui connaissent ce type de données numériques. Elle est donnée en détail au chapitre « Les syntaxes de Prolog III ».

© PrologIA

Le tuple vide

```
< >
```

Le double signe `<>` est un symbole conventionnel destiné à servir d'étiquette aux arbres qui, d'un point de vue sémantique, n'en ont pas. Ces arbres se réduisent donc à la suite de leurs fils : ils implantent en Prolog III la notion de suite finie d'arbres, qu'on appelle des *tuples*.

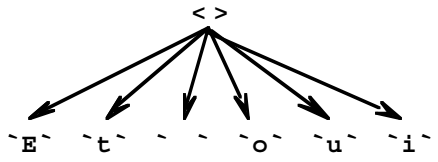
Comme nous le verrons, les tuples se notent $<a_1, \ldots a_n>$; le tuple vide se note donc tout naturellement `<>`. Ceci justifie le nom donné à ce signe.

Les chaînes de caractères

La notation courante des chaînes de caractères, à l'aide de guillemets :

```
"Et  oui"
```

est en réalité une deuxième notation pour l'arbre



En Prolog III, une chaîne de caractères est toujours structurée, puisque c'est un tuple de caractères. Les chaînes ne sont des entités atomiques qu'en tant qu'objets syntaxiques, et encore à la condition d'utiliser la notation des constantes-chaînes, `"Et oui"`.

Cas particulier, le tuple vide et la chaîne vide sont la même chose ; les deux notations `""` et `<>` sont donc équivalentes.

3. Opérations

Pour désigner des arbres qui ne sont pas des constantes vous devez écrire des formules qui combinent des constantes et des opérateurs, selon une syntaxe que nous indiquerons progressivement. Une telle formule exprime le résultat d'une *opération*. Une opération est définie sur un ensemble de *n-uplets* d'arbres ; à chacun elle associe un arbre[1] :

$$f : (a_1, \ldots a_n) \rightarrow f(a_1, \ldots a_n)$$

L'ensemble des *n*-uplets d'arbres pour lesquels une opération donnée est définie n'est pas obligatoirement égal à l'ensemble de tous les *n*-uplets ; on dit que l'opération est *partielle*. Spécifier le sous-ensemble des *n*-uplets sur lesquels l'opération est définie fait partie de la définition de l'opération en question.

On dispose de trois types d'opérations : les opérations booléennes, les opérations arithmétiques et les opérations de construction d'arbres. Avec une précision importante: *les opérations booléennes et arithmétiques ont leur signification mathématique habituelle.*

Cette remarque peut surprendre mais elle mérite d'être faite, notamment à l'intention des utilisateurs des Prolog précédents. Dans ces langages, des opérations notées `+`, `*`, `&`, etc… sont définies ou pourraient l'être ; cependant, elles n'y ont pas leur signification mathématique courante, elles ne sont que des opérations de construction d'arbres. Ainsi, en Prolog II, la formule `2 + 3` désigne l'arbre dont l'étiquette est `+` et les fils `2` et `3`. Un prédicat prédéfini peut ensuite interpréter cet arbre et en extraire le nombre `5`, mais cela est étranger au cœur de Prolog II, qui n'effectue aucun traitement spécifique des booléens ou des nombres.

1 Dans le jargon mathématique, une opération est donc une application
$f : D' \rightarrow D$, avec $D' \subset D^n$

© PrologIA

En Prolog III, au contraire, les opérations booléennes et arithmétiques sont connues en tant que telles du cœur du langage, qui leur donne leur signification logique ou arithmétique habituelle. Il en découle que l'arbre représenté par la formule `2 + 3` n'est autre que la feuille réduite au nombre `5`, résultat de l'opération indiquée.

Cette remarque aura encore plus d'intérêt quand nous considérerons les termes avec des variables. Comme pour l'addition précédente, la formule `2 + X` ne désignera pas un quelconque arbre non atomique mais bel et bien le nombre (inconnu) résultat de l'addition de `2` et du nombre inconnu `X`.

Opérations booléennes

Les opérations booléennes ne sont définies que si les opérandes sont des valeurs booléennes, c'est-à-dire des feuilles étiquetées par des constantes booléennes.

(1) L'opération unaire *non*

$a_1 \rightarrow$ `~` a_1

(2) L'opération binaire *et*

$(a_1, a_2) \rightarrow a_1$ `&` a_2

(3) L'opération binaire *ou* (non exclusif)

$(a_1, a_2) \rightarrow a_1$ `|` a_2

(4) L'opération binaire *implique*

$(a_1, a_2) \rightarrow a_1$ `=>` a_2

(5) L'opération binaire *équivalent*

$(a_1, a_2) \rightarrow a_1$ `<=>` a_2

Les résultats rendus par ces différentes opérations sont définis par le tableau suivant :

a_1	a_2	~a_2	a_1 & a_2	a_1 | a_2	a_1 => a_2	a_1 <=> a_2
0	0	1	0	0	1	1
0	1	0	0	1	1	0
1	0		0	1	0	0
1	1		1	1	1	1

Des termes corrects exprimant des opérations booléennes, ainsi que certaines fautes à ne pas commettre, sont montrés au § 5.

Opérations arithmétiques

Les opérations arithmétiques ne sont définies que si les opérandes sont des valeurs numériques, c'est-à-dire des feuilles étiquetées par des nombres.

Si aucun de ces nombres n'est flottant, alors l'opération est considérée comme une opération exacte sur des nombres rationnels. Si l'un au moins des opérandes est flottant, alors l'autre est transformé en flottant, sauf s'il l'était déjà, et l'opération est considérée comme étant l'opération correspondante sur les nombres flottants.

Les opérations arithmétiques sont :

(1) L'opération unaire “neutre”

$a_1 \rightarrow + a_1$

(2) L'opération unaire de *changement de signe*

$a_1 \rightarrow - a_1$

(3) L'opération binaire *addition*

$(a_1, a_2) \rightarrow a_1 + a_2$

(4) L'opération binaire *soustraction*

$(a_1, a_2) \rightarrow a_1 - a_2$

(5) L'opération binaire *multiplication*

© PrologIA

$(a_1, a_2) \rightarrow a_1 * a_2$

Afin d'alléger les écritures, la syntaxe de Prolog III permet d'écrire a_1a_2 à la place de a_1*a_2. A la condition, bien entendu, que cela ne change pas le sens de ce qui est écrit. Mais attention, il y a quelques pièges passablement vicieux : voyez le § 5.

(6) L'opération binaire *division*

$(a_1, a_2) \rightarrow a_1 / a_2$

La division n'est définie que si a_2 est différent de zéro.

Des termes corrects exprimant des opérations arithmétiques, ainsi que des fautes à ne pas commettre, sont montrés au § 5.

Opérations de construction d'arbres

Ces opérations permettent de construire des arbres non réduits à des feuilles. Nous avons :

(1) L'opération de construction d'arbre

$(a_1, a_2, \ldots a_n) \rightarrow a_1 (a_2, \ldots a_n)$

Pour que cette opération soit définie il faut que $n \geq 2$ et que a_1 soit une feuille. Elle a pour résultat l'arbre dont l'étiquette est[1] a_1 et dont les fils sont $a_2, \ldots a_n$.

(2) L'opération de construction de tuple

$(a_1, \ldots a_n) \rightarrow < a_1, \ldots a_n >$

[1] En toute rigueur il faudrait dire «l'arbre dont l'étiquette est l'étiquette de a_1», mais comme nous l'avons dit, Prolog III ne distingue pas une feuille de son étiquette.

Elle est définie quelque soit le n-uplet $(a_1, \ldots a_n)$, à la condition que $n \geq 1$; elle a pour résultat le *PIII-tuple* dont $a_1, \ldots a_n$ sont les éléments. On dit que n est la *longueur* du tuple construit. L'égalité suivante découle de la définition même des tuples :

$$<a_1, \ldots a_n> \;=\; <>(a_1, \ldots a_n)$$

(3) L'opération binaire de construction générale d'arbre

$(a_1, a_2) \rightarrow a_1\,[a_2]$

Cette opération n'est définie que si a_1 est une feuille et a_2 un tuple. Elle a pour résultat l'arbre qui a a_1 pour étiquette et a_2 pour suite de fils. Par définition, on a donc l'égalité :

$$a_1\,[<b_1, \ldots b_m>] \;=\; a_1\,(b_1, \ldots b_m)$$

(4) L'opération binaire de *concaténation de tuples*

$(a_1, a_2) \rightarrow a_1 \,.\, a_2$

Elle n'est définie que si a_1 et a_2 sont tous les deux des tuples. Elle a pour résultat le tuple dont les éléments sont ceux de a_1 suivis de ceux de a_2. Par définition, on a donc l'égalité :

$$<b_1, \ldots b_m>.<c_1, \ldots c_k> \;=\; <b_1, \ldots b_m, c_1, \ldots c_k>$$

© PrologIA

Les listes classiques

On peut aussi utiliser en Prolog III des *listes classiques*[1], basées sur la notion de paire pointée, comme elles existent en Lisp ou dans d'autres Prolog. Dans la syntaxe d'Edimbourg[2], ces listes s'écrivent de la manière suivante :

- liste vide :

  `[]`

- liste ayant les éléments e_1, e_2, … e_n et terminée par [] :

  `[` e_1`,` e_2`,` … e_n `]`

- liste ayant en tête les éléments e_1, e_2, … e_n et pour queue la liste q :

  `[` e_1 `,` e_2 `,` … e_n `|` q `]`

Il faut bien comprendre que ceci n'est pas une opération de construction d'arbres spécifiquement prise en compte par Prolog III, mais uniquement un autre moyen syntaxique d'exprimer certains arbres : la *paire pointée* et le célèbre *nil*. A cet effet, on donne un statut d'identificateur au double signe `[]` et on code le doublet de base `[`e_1`|`e_2`]` par l'arbre `[]`(`$e_1$`,`$e_2$`)`. Par conséquent, les expressions ci-dessus spécifient respectivement les arbres

`[]`

`[]`(`$e_1$`,[]`(`e_2`, … []`(`$e_n$`,[]) …))`

`[]`(`$e_1$`,[]`(`e_2`, … []`(`$e_n$`,`$q$`) …))`

Même si les tuples de Prolog III sont une amélioration indiscutable des listes classiques, toutes les règles prédéfinies de Prolog III qui sont héritées de Prolog II n'utilisent que ces dernières, afin de ne pas changer de comportement.

---

1 Nous disons ici “classiques” par opposition aux “nouvelles listes” que sont les tuples, mais dans la suite de cette documentation nous appellerons les listes classiques tout simplement des “listes”ou bien encore des "listes binaires".

2 Ces listes ne peuvent pas être exprimées en Prolog III avec la syntaxe qu'elles ont en Prolog II, c'est-à-dire « $a_1.a_2.\ \ldots\ .\ a_n$ », le point ayant ici un tout autre rôle syntaxique (il exprime la concaténation)

4. Variables

Les variables ne sont pas d'autres éléments du domaine de Prolog III, dont on n'aurait pas encore parlé, et qui existeraient en plus des arbres. Essentiellement, les variables sont une autre manière de représenter les arbres, ainsi qu'il découlera des explications suivantes.

Du point de vue de la syntaxe, le principal signe de reconnaissance d'une variable est d'avoir pour nom une suite de lettres, de chiffres et des deux caractères apostrophe (') et blanc souligné (_), commençant par une lettre et *dont le second caractère, s'il existe, n'est pas une lettre*. Les formules suivantes sont des noms de variables[1]

```
X
Y'
Z2
V_ariable
```

Il y a une profonde différence entre ce que signifient les variables dans la plupart des langages classiques et ce qu'elles représentent en Prolog III. Dans les langages classiques, une valeur est attachée à chaque variable. L'exécution d'un programme produit toutes sortes de modifications de ces valeurs et obtient la solution du problème sous forme de valeurs finales des variables. Dans ces langages, une variable désigne constamment un objet parfaitement connu, ou bien l'utilisation de la variable est illégale.

EN PROLOG III, LES VARIABLES REPRéSENTENT DES ARBRES INCONNUS.

A l'opposé, une variable Prolog III désigne un arbre inconnu, exactement comme une inconnue dans une équation mathématique désigne un nombre inconnu, par exemple **x** dans

---

1 Dans la syntaxe marseillaise de Prolog III, les variables n'ont pas à commencer par une lettre majuscule, ni les identificateurs par une lettre minuscule. Cependant, telle est la règle dans la syntaxe d'Edimbourg. Pour cette raison, bien qu'écrits dans la syntaxe originale, la plupart des expressions Prolog III données en exemple dans le texte seront, autant que possible, compatibles avec les deux syntaxes.

© PrologIA

```
X = (1/2)X + 1
```

Dans cet esprit, le déroulement d'un programme Prolog III ne vise pas à *modifier* les valeurs des variables, mais à les *déterminer*. Dès leur entrée en scène les variables pourront apparaître dans des termes et des contraintes, comme `X < 10` ; justement, programmer en Prolog III ne sera parfois rien d'autre qu'écrire les contraintes qui pèsent sur un arbre initialement inconnu, représenté par une variable. Le travail de Prolog III sera alors de résoudre ces contraintes et de dévoiler l'arbre en question.

5. Termes

Ayant introduit les constantes, les opérations et les variables, nous disposons de tout ce qu'il nous faut pour définir les termes dans toute leur généralité. Comme nous l'avons dit, *les termes sont les formules qui représentent les arbres dans les programmes*.

Voici les plus significatives parmi les expressions BNF qui définissent la syntaxe des termes. La totalité de ces expressions est donnée au chapitre « Les syntaxes de Prolog III ».

Syntaxe

<terme>
::= <terme$_1$>
::= <terme> **=>** <terme$_1$>
::= <terme> **<=>** <terme$_1$>

<terme$_1$>
::= <terme$_2$>
::= **+** <terme$_2$>
::= **-** <terme$_2$>
::= <terme$_1$> **+** <terme$_2$>
::= <terme$_1$> **-** <terme$_2$>
::= <terme$_1$> | <terme$_2$>

<terme$_2$>
::= <terme$_3$>
::= <terme$_2$> **/** <terme$_3$>
::= <terme$_2$> ***** <terme$_3$>
::= <multiplicande> <variable>
::= <terme$_2$> **&** <terme$_3$>

<terme$_3$>
::= <terme$_4$>
::= **~** <terme$_3$>

<terme$_4$>
::= <terme$_5$>
::= <terme$_5$> **(** <suite de termes> **)**
::= <terme$_5$> **[** <terme> **]**
::= <terme$_4$> **.** <terme$_5$>

<terme$_5$>
::= <variable>
::= <constante>
::= **<** <suite de termes> **>**
::= **(** <terme> **)**

<multiplicande>
::= <entier>
::= <réel>
::= **(** <terme> **)**

<suite de termes>
::= <terme>
::= <terme> **,** <suite de termes>

© PrologIA

Un point important apparaît immédiatement : les expressions qui nous ont servi à définir les arbres et les opérations sur les arbres sont des termes. En fait, pour expliquer les constantes et les opérations nous n'avons pas pu faire autrement que de définir implicitement la syntaxe des *termes sans variable*. Considérons ces formules comme étant acquises : ce que nous leur ajoutons maintenant est la possibilité d'y mettre des variables. Plus précisément, nous permettons désormais d'*écrire une variable à tout endroit d'un terme-sans-variable où pouvait figurer l'expression d'un arbre*.

Limitations

La syntaxe précédente autorise donc l'écriture de toute sorte de termes. En particulier, elle n'établit pas de dissymétrie entre les deux opérandes des opérations arithmétiques, booléennes ou de construction d'arbre. Il faut savoir qu'au niveau des mécanismes de base de Prolog III il existe deux importantes restrictions.

La première concerne les termes chapeautés par un opérateur arithmétique : *les expressions arithmétiques doivent être linéaires*

Cela signifie que dans une multiplication a_1 `*` a_2 un des deux opérandes ne doit pas contenir de variable, et que dans une division a_1 `/` a_2 le deuxième opérande, a_2, ne doit pas contenir de variable. Cela interdit des termes comme `(2x+3)*(4y+5)` ou `x/y`.

La deuxième restriction concerne l'opération de concaténation : *dans une concaténation, la longueur de l'opérande gauche doit être connue.*

Cela interdit l'écriture de termes comme `x.y`, sauf si une contrainte, comme `x = <z1,z2,z3>` ou `x::3` (les contraintes sont expliquées au §7) impose à `x` de posséder une longueur déterminée, ici 3.

Bien que ces limitations soient fondamentales au niveau des mécanismes de base de Prolog III, dans la plupart des situations pratiques vous pourrez ne pas en tenir compte, un dispositif de retardement automatique se chargeant à votre place de “geler” l'expression conflictuelle jusqu'à ce qu'assez de variables soient connues pour que l'expression puisse être considérée comme légitime. Ce mécanisme est décrit au chapitre « Retardements ».

Exemples

Voici une collection de termes corrects et d'expressions erronées, illustrant la syntaxe des termes et justifiant un certain nombre de compléments d'information et de mises en garde à leur sujet.

Identificateurs

```
pomme                                    (1)
pomme'                                   (2)
nb_fils                                  (3)
Paul_Hochon                              (4)
zZ44a_45b_c46'                           (5)
```

Il s'agit d'identificateurs corrects pour la syntaxe marseillaise. On notera que pour la syntaxe d'Edimbourg (expliquée au chapitre « Les syntaxes de Prolog III »), notre exemple *(4)* n'est pas un identificateur mais une variable, car il commence par une lettre majuscule.

© PrologIA

Identificateurs incorrects[1]

```
t_iti                                                    (1)
i'm_happy                                                (2)
nb-freres                                                (3)
1dou                                                     (4)
```

Les exemples *(1)* et *(2)* ne sont pas des identificateurs, car leur deuxième caractère n'est pas une lettre ; ce sont des noms de variables corrects. Les deux exemples suivants *(3)* et *(4)* sont illégaux : le signe moins n'est pas permis à l'intérieur d'un mot, un identificateur ne peut pas commencer par un chiffre.

Entiers

```
2500                                                     (1)
81591528324789773434561126959611589427200000 0          (2)
012                                                      (3)
00                                                       (4)
```

Entiers incorrects

```
2 500                                                    (1)
8.0                                                      (2)
0.                                                       (3)
```

La première expression ne définit pas un entier, mais deux entiers séparés par un blanc. Les deux expressions suivantes définissent des nombres flottants légitimes, mais non des entiers corrects.

[1] Il faudrait dire « des expressions qui ne sont pas des identificateurs corrects », bien sur !

Flottants

```
1.e6                    ( 1 )
.5e6                    ( 2 )
31.4e-1                 ( 3 )
4.                      ( 4 )
.666                    ( 5 )
```

Ces expressions définissent respectivement les nombres 10^6 ; $0{,}5 \times 10^6$; $31{,}4 \times 10^{-1}$; 4 ; 0,666

Flottants incorrects

```
e10                     ( 1 )
.e10                    ( 2 )
1.2e                    ( 3 )
3e10                    ( 4 )
```

L'exemple *(1)* ne spécifie pas un nombre, mais une variable. Les autres exemples nous rappellent que : la mantisse doit avoir au moins un chiffre *(2)*, l'exposant ne peut être omis *(3)* pas plus que le point décimal *(4)*.

Caractères

```
`A`                     ( 1 )
`\x41`                  ( 2 )
`\101`                  ( 3 )
`\41`                   ( 4 )
`\n`                    ( 5 )
```

Les trois premiers exemples spécifient le même caractère : la lettre `A`, dont le code ASCII est `41` en hexadécimal, ou `101` en octal. On notera que si le zéro a été omis dans le quatrième exemple, qui spécifie le caractère `!` (041 en octal), c'est parce qu'il n'y a aucune ambiguïté au sujet du caractère immédiatement après l'expression `\41`. L'exemple *(5)* indique le caractère de fin de ligne.

© PrologIA

Chaînes de caractères

```
"Pinocchio va a l'ecole"                          (1)
"Pinocchio va \
a l'ecole"                                        (2)
"Cette chaine contient\nun retour chariot"        (3)
"Les crochets \"[\" et \"]\" indiquent…"          (4)
""                                                (5)
"A\tB"                                            (6)
"A\x09B"                                          (7)
"A\x7B"                                           (8)
```

Les chaînes *(1)* et *(2)* sont identiques. La chaîne *(4)* a pour contenu :

```
Les crochets "[" et "]" indiquent…
```

La chaîne *(5)* est la chaîne vide. Elle peut aussi s'écrire

```
< >
```

Les chaînes *(6)*,et *(7)* sont identiques (A et B séparés par une tabulation horizontale).

L'exemple *(8)* représente la chaine

```
A{
```

Chaînes de caractères incorrectes

```
"""                                               (1)
"Les crochets ""["" et…"                          (2)
"11\1112"                                         (3)
```

L'exemple *(1)* pose le problème bien connu de la présence dans une chaîne du délimiteur des chaînes lui-même. L'exemple *(2)* montre une mauvaise manière de résoudre ce problème. L'exemple *(3)*, à comparer avec l'exemple *(8)* des chaînes correctes, montre un cas où l'on ne peut se dispenser d'écrire les trois chiffres du code ASCII en octal du caractère. L'écriture correcte de cette chaîne serait probablement

```
"11\01112"   (ou   "11\t12")
```

Expressions numériques

Appelons expression numérique un terme constitué par l'application d'un opérateur arithmétique à un ou plusieurs opérandes. En voici des exemples :

```
-X                                          (1)
+X                                          (2)
355/113                                     (3)
4 + 1/2 + 0.333                             (4)
X + 3Y + 5Z/4 + (14+9/17)T                  (5)
(1000 - 2*3*4*5)Y                           (6)
(1/2 + 2/3 + 3/4 + 4/5)*(X + Y - 2T/5)      (7)
(X + Y)*(X - Y)                             (8)
(2T/5 + X - Y)/(X + Y - 1/2)                (9)
```

L'expression *(1)* (resp. *(2)*) indique l'opération de changement de signe (resp. l'opération "neutre") appliquée au nombre inconnu `X`. L'une et l'autre n'ont de sens que pour des nombres ; elles imposent donc à `X` de représenter une valeur numérique. En outre, leur résultat est par définition un nombre. Comme les autres opérations arithmétiques, elles ont donc bien plus de conséquences que la simple transformation de la valeur de leur opérande : ils imposent des contraintes de type à leurs opérandes et aux expressions dans lesquelles ils apparaissent. La notion de contrainte est expliquée en détail au § 7.

L'exemple *(4)* montre divers cas où la syntaxe de Prolog III permet d'omettre l'opérateur `*`. Une écriture équivalente de cette expression est

```
X + 3*Y + 5*Z/4 + (14+9/17)*T
```

Les exemples *(8)* et *(9)* sont des expressions non linéaires, un dispositif de retardement automatique est donc mis en place. On se référera pour plus de détails sur ce sujet au chapitre « Retardements ».

© PrologIA

Expressions numériques incorrectes

```
5(X + Y)                                    ( 1 )
4/5X                                        ( 2 )
X5                                          ( 3 )
XY                                          ( 4 )
3*-X                                        ( 5 )
```

En tant qu'expression arithmétique `5(X + Y)` est incorrect. Cependant, ce terme est l'écriture correcte d'un arbre ayant `5` pour étiquette et le nombre résultat de l'addition `X + Y` pour unique fils.

L'exemple `4/5X` est syntaxiquement incorrect. Pour indiquer les quatre cinquièmes de la valeur représentée par `X` il faut écrire `(4/5)X`.

L'expression `X5` n'est pas incorrecte, mais elle ne représente pas une expression arithmétique, uniquement une variable ayant ce nom. De la même manière, `XY` est un identificateur, non un produit.

Enfin, l'expression `3*-X` doit être réécrite `3*(-X)`

Expressions booléennes

```
1'                                          ( 1 )
X | 0'                                      ( 2 )
X | ~Y                                      ( 3 )
~~Y                                         ( 4 )
X & X                                       ( 5 )
X & 1'                                      ( 6 )
(A & B) => (C & D)                          ( 7 )
(U3 & ( ~X1 <=> X3 )) | 0'                  ( 8 )
```

Expressions “mixtes” arithmético-booléennes

La règle est simple : *elles sont toutes incorrectes*. Autrement dit, il n'y a pas en Prolog III d'expression arithmétique correcte comprenant des sous-expressions booléennes, pas plus qu'il n'y a d'expression booléenne correcte comprenant des sous-expressions arithmétiques. Cela découle de la définition des opérations arithmétiques et des opérations booléennes. Les formules suivantes sont donc toutes incorrectes :

```
-1'                                    (1)
(X & Y) + Z                            (2)
~0                                     (3)
X & (Y + Z)                            (4)
```

On accepte sans grande difficulté ces limitations qui, somme toute, appartiennent à tous les langages “propres”. On peut s'accommoder moins facilement de la restriction suivante : *en Prolog III il n'y a aucun opérateur fournissant un résultat booléen découlant de propriétés des nombres*. Les opérateurs qui s'appliquent à des nombres donnent tous des nombres pour résultat, tandis que ceux qui rendent des booléens n'ont de sens qu'appliqués à des booléens.

Des expressions comme la suivante sont donc fondamentalement incorrectes :

```
(0 < X) & (X < 9)
```

Mais alors, quel est le statut en Prolog III de l'égalité, l'inégalité, les comparaisons numériques, etc…? Il ne faut pas oublier ceci : `=`, `#`, `<`, `<=`, `>`, `>=` *ne sont pas des opérateurs*. Ces symboles servent à exprimer des *relations*, lesquelles servent à écrire des *contraintes*. Le propre d'une contrainte est d'être vérifiée ou non, et en pratique il nous arrivera de dire qu'elle est *vraie* ou *fausse*. Mais il ne s'agit pas là des valeurs booléennes du domaine de Prolog III, `0'` et `1'`. Une contrainte n'est pas un terme, et ne peut pas être vue comme exprimant une opération.

© PrologIA

Autres termes

```
mesure(pierre, <180/100, metres>, 1')          (1)
mesure[<pierre, <180/100, metres>, 1'>]        (2)

<mesure, pierre, <180/100, metres>, 1'>        (3)
<>(mesure, pierre, <180/100, metres>, 1')      (4)

4(X + Y)                                       (5)

(4z+y)(caractere(`c`, "caractere `c`"))        (6)
```

On notera qu'en Prolog III les termes *(1)* et *(3)* ne sont pas équivalents[1]. En revanche, *(1)* et *(2)* le sont, ainsi que *(3)* et *(4)*. L'exemple *(5)* montre un terme correct, qui n'est pas le nombre 4 × (x + y) mais l'arbre ayant 4 pour étiquette et le nombre (x + y) pour unique fils.

On notera aussi que l'étiquette du dernier exemple a dû être parenthésée.

Termes incorrects

```
triplette (X,Y,Z)                              (1)
ff(X,Y)(Z,T)                                   (2)
uneFeuille()                                   (3)
```

L'exemple *(1)* rappelle un des (rares) endroits où la syntaxe ne tolère pas l'insertion d'un blanc : entre l'étiquette d'un arbre et la parenthèses ouvrante qui précède les fils.

Du seul point de vue de la syntaxe, l'exemple *(2)* pourrait être corrigé en l'écrivant `(ff(X,Y))(Z,T)`. Mais ce terme serait alors sémantiquement incorrect, puisque l'étiquette d'un arbre doit être une feuille.

---

1 Puisqu'il ne s'agit dans cet exemple que de termes sans variable, «être équivalent» signifie tout simplement «représenter le même arbre»

L'exemple *(3)* montre que dans le cas d'une feuille, la syntaxe interdit l'écriture d'une paire de parenthèses encadrant la suite (vide) des fils. Ceci est tout-à-fait cohérent avec la propriété que Prolog III ne distingue pas une feuille de son étiquette.

Que représentent les termes ?

Les termes représentent les arbres dans les programmes Prolog III. On peut comprendre cela de deux manières :

• *Un terme désigne un arbre partiellement inconnu*. Nous avons déjà expliqué qu'une variable, terme particulier, représentait un arbre inconnu dont la détermination était l'objet du programme Prolog III considéré. De manière analogue, un terme représente dans le cas général un arbre dont certains constituants seulement sont connus, l'objet du programme étant de dévoiler les autres constituants. Les parties inconnues sont nécessairement des sous-arbres ; les parties connues peuvent être des sous-arbres ou bien des éléments qui définissent le squelette de l'arbre. Par exemple, le terme

```
< P + L, 2P + 4L >
```

représente un tuple dont le squelette est connu (c'est un tuple de longueur deux) alors que les sous-arbres qui le constituent sont les nombres inconnus résultats des opérations `P + L` et `2P + 4L`.

• *Un terme représente un ensemble d'arbres*, à savoir l'ensemble obtenu en donnant toutes les valeurs possibles aux variables qu'il contient. Ainsi, le terme donné en exemple ci-dessus représente l'ensemble infini de tous les tuples de deux nombres. Cet ensemble est identique à celui défini par le terme `< +X, +Y >` et n'a aucun élément en commun avec les ensembles définis par les tuples `< X >` ou `< X, Y, Z >`

Nous retrouvons bien entendu les deux cas extrêmes des termes : un terme réduit à une variable représente l'ensemble de tous les arbres, tandis qu'un terme sans variable représente l'ensemble réduit à l'unique arbre dont le terme en question est l'expression écrite.

© PrologIA

Affectation

Attardons-nous un instant sur le concept *« donner des valeurs aux variables d'un terme »* pour un peu mieux le formaliser ; cela nous servira plus tard à comprendre ce qu'est la résolution d'un système de contraintes. Nous avons déjà dit qu'une variable pouvait représenter n'importe quel élément du domaine de Prolog III ; définir une *affectation* d'un ensemble de variables c'est tout simplement choisir une valeur pour chacune des variables en question. Si l'ensemble est

$$V = \{ x_1, x_2, \ldots x_n \}$$

alors une affectation de cet ensemble sera un ensemble de couples (x_i, a_i), notés plutôt $x_i \leftarrow a_i$, comme

$$A = \{ x_1 \leftarrow a_1, x_2 \leftarrow a_2, \ldots x_n \leftarrow a_n \}$$

signifiant : l'arbre a_1 est donné pour valeur à la variable x_1, l'arbre a_2 à la variable x_2, etc… Une affectation n'est donc rien d'autre qu'une application, définie en extension (élément par élément), d'un ensemble de variables dans le domaine de Prolog III.

Soit t un terme ; chaque affectation A de l'ensemble des variables figurant dans t permet de transformer ce dernier en un arbre $a = t_{/A}$. Grosso modo, il suffit de *« remplacer chaque variable de t par sa valeur dans A »*. Si on voulait être plus rigoureux, il faudrait définir $t_{/A}$ pour chaque modèle de terme t, de la manière suivante :

- si t se réduit à une constante k, alors a est l'arbre k
- si t se réduit à une variable x, alors a est l'arbre donné pour valeur à x dans l'affectation A
- si t est de la forme $< t_1, t_2, \ldots t_n >$ et si $a_1, a_2, \ldots a_n$ sont les arbres $t_{1/A}$, $t_{2/A}$, $\ldots t_{n/A}$ alors a est l'arbre $< a_1, a_2, \ldots a_n >$
- si t est de la forme $t_1 (t_2, \ldots t_n)$ et si $a_1, a_2, \ldots a_n$ sont les arbres $t_{1/A}$, $t_{2/A}$, $\ldots t_{n/A}$ alors
  - si a_1 est une feuille, alors a est l'arbre $a_1 (a_2, \ldots a_n)$
  - sinon, a est *indéfini*

- si t est de la forme t_1 `[`t_2`]` et si a_1 et a_2 sont les arbres $t_{1/A}$ et $t_{2/A}$ alors
  - si a_1 est une feuille et a_2 un tuple, alors a est l'arbre a_1 `[`a_2`]`
  - sinon, a est *indéfini*

etc…

Par exemple, l'affectation { `P ← 2, L ← 3` } permet de transformer le terme `< P + L, 2P + 4L >` en l'arbre `< 2 + 3, 2*2 + 4*3 >` c'est-à-dire `< 5, 16 >`.

Des termes particulièrement subtils…

Pour en finir avec les termes, remarquons que la prise en considération spécifique des opérations arithmétiques et booléennes par le cœur de Prolog III a des conséquences dont la finesse mérite d'être soulignée. Examinons par exemple les deux termes

```
X
X + 0
```

Le premier désigne l'ensemble de tous les arbres : tout élément du domaine de Prolog III peut être représenté dans un programme par la variable `X`. Le second comporte des éléments connus : l'opération d'addition `+` et la constante `0`. Ce terme représente comme nous l'avons dit l'ensemble des arbres qu'on peut obtenir en donnant des valeurs à la variable `X`. Or l'addition n'est définie que sur les valeurs numériques, c'est-à-dire des arbres réduits à une feuille étiquetée par un nombre, et elle rend pour résultat des valeurs numériques ; d'autre part, tout nombre y peut s'écrire sous la forme $y = x + 0$. Par conséquent, le terme `X + 0` définit le sous-ensemble du domaine de Prolog III constitué par tous les nombres.

On peut désigner ce même sous-ensemble par le terme

```
1*X
```

ou encore de manière plus “économique”, par le terme

© PrologIA

```
+X
```

en effet, l'opération arithmétique + unaire impose à son opérande d'être un nombre, et rend ce nombre comme résultat. Remarquons au passage que, bien que qualifiée d'opération *neutre*, il serait bien trompeur de prendre le + unaire pour une opération sans effet.

Bien entendu, toutes ces considérations s'appliquent de manière analogue aux expressions booléennes. Ainsi, les termes

```
X & 1'
X | 0'
~~X
```

sont trois manières de désigner l'ensemble de tous les booléens. Si l'on préfère, on peut considérer que ces trois expressions représentent un arbre qui est réduit à une feuille étiquetée par un booléen et qui, pour le reste, est inconnu.

6. Relations

En Prolog III les relations unaires et binaires traduisent des conditions portant sur des arbres, comme « `<aa,bb,cc>` *est différent de* `aa(bb,cc)` » *ou* « `1` *est supérieur à* `0` ». Nous disposons de relations tout-à-fait générales, comme l'égalité entre arbres, et de relations spécifiques qui n'ont de sens que pour une catégorie bien particulière d'arbres, comme les relations d'ordre sur les nombres ou l'implication entre booléens.

Les relations ne sont pas des opérations[1]. Le propre d'une opération, appliquée à un *n*-uplet d'arbres, est de rendre un arbre à titre de résultat ; le propre d'une relation, appliquée à un *n*-uplet d'arbres, est d'être ou non vérifiée. Bien que définie sur des arbres ou des couples d'arbres, elle ne leur associe pas des éléments du domaine de Prolog III et, même si dans le langage courant un dit qu'une relation est « vraie » ou « fausse », il ne faut pas voir cela comme des occurrences des valeurs booléennes `1'` et `0'` du domaine de Prolog III.

Comme les opérations, les relations sont *partielles*. L'ensemble des arbres ou des couples d'arbres pour lesquels une relation est définie n'est pas nécessairement égal à l'ensemble de tous les arbres ou de tous les couples d'arbres.

Les relations binaires sont au nombre de sept :

Egalité et inégalité

La condition

a_1 `=` a_2

se lit « les arbres a_1 et a_2 sont égaux ».

La condition

a_1 `#` a_2

se lit « les arbres a_1 et a_2 ne sont pas égaux ».

---

1 Rappelons que d'un point de vue mathématique, définir une relation binaire sur un ensemble D c'est choisir un sous-ensemble B de l'ensemble D × D des couples d'éléments de D ; les couples appartenant à B sont dits *vérifier la relation*. De la même manière, définir une relation unaire sur D c'est choisir le sous-ensemble U de D des éléments pour lesquels la relation est vérifiée.

© PrologIA

L'égalité des arbres est une relation qui se définit récursivement en affirmant que deux arbres a_1 et a_2 sont égaux si

- les étiquettes de a_1 et a_2 sont *égales-au-sens-des-étiquettes*
- a_1 et a_2 ont le même nombre n de fils
- (si $n \neq 0$) le premier fils de a_1 est égal au premier fils de a_2, le deuxième fils de a_1 est égal au deuxième fils de a_2, etc… jusqu'au n-ème fils de a_1 qui est égal au n-ème fils de a_2.

Bien entendu, ceci suppose que l'on a préalablement défini une *égalité-des-étiquettes*. Pour que deux étiquettes soient égales, il faut

- qu'elles soient de même type (deux identificateurs, deux nombres, deux caractères etc…)
- qu'elles soient égales pour les critères d'égalité propres à leur type.

Pour l'essentiel, ces critères sont les suivants :

- *identificateurs* : deux identificateurs sont égaux si leurs formes non abrégées s'épellent exactement de la même manière. On notera bien qu'il n'y a pas d'équivalence entre les lettres majuscules et les minuscules. La “forme non abrégée” des identificateurs est expliquée au chapitre « Le contrôle et l'environnement des programmes ».
- *caractères* : deux *caractères imprimables internationaux* sont égaux si leur forme écrite est la même, ce qui équivaut à dire qu'ils ont la même représentation interne (code ASCII). Pour les caractères nationaux ou les caractères non imprimables il y a lieu de se référer à la documentation particulière du système Prolog III utilisé.
- *nombres rationnels*. L'égalité entre nombres entiers est une notion primitive, héritée des mathématiques, qui ne pose aucun problème. En ce qui concerne les nombres fractionnaires, si l'on sait que Prolog III les conserve en précision infinie et sous leur forme irréductible, là encore l'égalité. ne pose aucun problème.

- *nombres flottants*. L'égalité entre deux nombres flottants `x` et `y`, ou entre deux nombres `x` et `y` dont l'un au moins est flottant, est définie par une relation de la forme `|x - y| ≤` ε, où ε est un seuil qui prend en compte l'ordre de grandeur de `x` et `y`, les particularités de la représentation des nombres flottants dans la machine utilisée et éventuellement une estimation de l'erreur commise dans l'expression où `x` et `y` apparaissent.
- les cas restants, c'est à dire les deux booléens `0'` et `1'`, et le tuple vide `<>` sont sans mystère : chacune de ces valeurs est différente des autres.

Implication

La condition

a_1 `=>` a_2

s'énonce : « les arbres a_1 et a_2 sont tous les deux des booléens et si a_1 vaut `1'` alors a_2 vaut `1'` ». Puisque a_1 et a_2 sont des booléens, il n'existe que quatre cas possibles et nous pouvons définir cette contrainte par un tableau :

a_1	a_2	*la contrainte* a_1 `=>` a_2 *est-elle satisfaite ?*
`0'`	`0'`	*oui*
`0'`	`1'`	*oui*
`1'`	`0'`	*non*
`1'`	`1'`	*oui*

Comparaisons numériques

Les conditions

a_1 `<` a_2
a_1 `<=` a_2
a_1 `>` a_2
a_1 `>=` a_2

© PrologIA

sont vérifiées si les arbres a_1 et a_2 sont tous les deux des nombres et si a_1 est inférieur (resp. inférieur ou égal, supérieur, supérieur ou égal) à a_2

Relations unaires

Les relations unaires sont souvent appelées des relations de type, car elles concernent principalement la *nature de l'étiquette* du nœud initial de l'arbre sur lequel elles portent. Dans certains cas, elles mettent aussi en jeu le *nombre de fils* du nœud initial.

```
a !numt
a !chart
a !boolt
a !idt
```

Ces conditions sont vérifiées si le nœud initial de l'arbre *a* est étiqueté par, respectivement, un nombre, un caractère, un booléen, un identificateur. Aucune condition n'est imposée pour les autres nœuds.

```
a !tuple
```

Cette condition est vérifiée si le nœud initial de l'arbre *a* est `<>`, autrement dit si *a* est un PIII-tuple. Aucune condition ne doit être vérifiée par les autres nœuds.

```
a ::n
```

Cette condition se lit « *n* est un nombre entier non négatif et du nœud initial de *a* partent *n* branches ». Souvent mise en conjonction avec la précédente, elles équivalent alors, ensemble, à « *a* est un tuple de longueur *n* »

```
a !num
a !char
a !bool
a !id
```

Ces conditions sont vérifiées si *a* est une feuille, et si son unique nœud est étiqueté par un nombre (resp. un caractère, un booléen, un identificateur). Par conséquent :

a `!num`	équivaut à	*a* `!numt` et *a* `::0`
a `!char`	équivaut à	*a* `!chart` et *a* `::0`
a `!bool`	équivaut à	*a* `!boolt` et *a* `::0`
a `!id`	équivaut à	*a* `!idt` et *a* `::0`

7. Contraintes

De la même manière que nous sommes passés des arbres aux termes, nous allons passer des conditions aux contraintes. Une condition était constituée par l'application d'une relation, entité mathématique, à un arbre ou un couple d'arbres, éléments du domaine de Prolog III. Une contrainte sera un objet syntaxique, formé à partir du symbole qui exprime une relation et un terme ou un couple de termes, sur lesquels on dira encore que la relation porte. Nous allons donc

- nous transporter sur un plan syntaxique[1]
- introduire les variables

Syntaxe

Dans les programmes Prolog III, les contraintes sont toujours exprimées au sein de *systèmes de contraintes*. Un système de contraintes est une suite finie de contraintes séparées par des virgules, encadrée par des accolades :

<système de contraintes>
::= { }
::= { <contrainte> {, <contrainte>} }

[1] En expliquant les conditions sur des arbres nous n'avons pas pu faire autrement que de définir en même temps les symboles qui expriment les relations ; nous avons donc déjà donné l'essentiel de la syntaxe des contraintes. Bien entendu, une contrainte sur des termes sans variable n'est rien d'autre que l'expression écrite d'une condition sur des arbres.

© PrologIA

<contrainte>
::= <terme> **=** <terme>
::= <terme> **#** <terme>
::= <terme> **=>** <terme>
::= <contrainte de typage>
::= <contrainte Sup>
::= <contrainte Inf>

<contrainte Sup>
::= <terme> <sup> <terme>
::= <contrainte Sup> <sup> <terme>

<contrainte Inf>
::= <terme> <inf> <terme>
::= <contrainte Inf> <inf> <terme>

<inf>
::= **<** | **<=**

<sup>
::= **>** | **>=**

<type d'arbre>
::= **!idt** | **!boolt** | **!numt** | **!chart** | **!tuple**
::= **!id** | **!bool** | **!num** | **!char**

<contrainte de typage>
::= <terme> <type d'arbre>
::= <terme> **::** <terme>

Relativement à ce que nous savions déjà, cette syntaxe ajoute surtout un détail : il est permis d'agglutiner plusieurs contraintes <, ≤, > ou ≥ à condition qu'elles soient toutes de type <, ≤ ou toutes de type >, ≥. On pourra donc écrire **{ 1 < x < y ≤3 }** à la place de **{ 1 < x, x < y, y ≤ 3 }**

Exemples

Voici des contraintes syntaxiquement correctes :

```
X  !num                                    (1)
F  !id                                     (2)
A  #  B(I,  J  +  1)                       (3)
<P,  "Durand">  =  <"Pierre",  N>          (4)
A  =>  B  &  C                             (5)
0  <  T  <  3X/4  +  5Y                    (6)
```

et des formules qui ne sont pas des contraintes correctes :

```
4X  +  Z                                   (1)
TT(X)                                      (2)
A  &  B                                    (3)
(X  =  Y)  &  (Y  =  Z)                    (4)
```

(1), *(2)* et *(3)* souffrent du même mal : ce sont des termes corrects, non des contraintes. Pour *(4)* c'est plus grave : une contrainte (ici `X = Y`) ne peut en aucun cas figurer dans un terme, de plus *(4)* traduit une confusion entre les booléens, éléments du domaine de Prolog III, et la vérité ou la fausseté des contraintes.

8. Résolution des systèmes de contraintes

Une condition portant sur des arbres est généralement la formalisation d'un énoncé dont implicitement on prétend qu'il est vrai, comme *« a est égal à b »*. On peut en dire autant au sujet d'une contrainte portant sur des termes sans variable, puisque de tels termes représentent sans ambiguïté des arbres uniques. Il nous reste à définir ce que signifie « être vérifiée » pour une contrainte portant sur des termes avec des variables.

© PrologIA

Nous avons expliqué comment une affectation de l'ensemble des variables permet de transformer un terme en un arbre ; appliqué à chacun des termes qui y figurent, le même procédé permet de transformer une contrainte sur des termes en une condition sur des arbres. Un système de contraintes sera dit “vérifiable”, ou plutôt *soluble*, s'il existe au moins une affectation des variables apparaissant dans le système par laquelle ce dernier se transforme en un ensemble de conditions vraies sur des arbres. L'affectation en question s'appelle une *solution* du système de contraintes.

Par exemple, le système de contraintes

```
{ P ≥ 0, L ≥ 0, P + L = 16 }
```

est soluble puisque, parmi d'autres, l'affectation

```
{ P ← 8, L ← 8 }
```

le transforme dans l'ensemble de conditions :

```
{ 8 ≥ 0, 8 ≥ 0, 8 + 8 = 16 }
```

Parlant familièrement nous dirons que, où qu'elles apparaissent dans un programme Prolog III, les contraintes “demandent” à être vérifiées. Une grande partie du travail de la machine Prolog III sera de veiller à ce que le système de toutes les contraintes prises en considération soit constamment soluble. Mais on ne se contentera pas de cela : la solubilité étant acquise, les valeurs qu'on doit affecter aux variables du système de contraintes, ou à certaines de ces variables, pour le transformer en un ensemble de conditions vraies nous intéressent beaucoup ; l'obtention de ces valeurs était souvent le but principal du programme en question.

Par exemple, le système de contraintes

```
{ P ≥ 0, L ≥ 0, P + L = 16, 2P + 4L = 44 }
```

est soluble lui aussi, puisque l'affectation

```
{ P ← 10, L ← 6 }
```

le transforme en un ensemble de conditions vraies. Mais celle-ci est maintenant l'unique affectation ayant cette propriété ; les valeurs qui y figurent constituent donc la solution *unique* du système, ce qui sans doute intéresse beaucoup l'auteur de ce programme.

Les contraintes sont écrites avec des relations partielles et des termes qui eux-mêmes sont construits à l'aide d'opérations partielles. Lorsqu'on considère une affectation qui transforme une contrainte en une condition sur des arbres, avant de pouvoir se demander si la condition est satisfaite, il faut déjà que les arbres qui y figureront soient bien définis. Pour cette raison les contraintes de Prolog III imposent généralement aux variables qui y figurent bien plus de limitations que celles qu'exprime au premier chef la relation qui chapeaute la contrainte.

Par exemple, la contrainte

```
X = +Y
```

fait intervenir l'opération « + unaire », qui n'est définie que sur les nombres. Toute affectation de `X` et `Y` qui transformera cette contrainte en une condition vraie devra affecter un nombre à `Y`. Cette contrainte impose d'abord à la variable `Y` d'être numérique, ensuite d'être égale à la variable `X`. Autrement dit, les systèmes { `X = +Y` } et { `Y !num, X = Y` } sont équivalents[1]. De manière analogue, les systèmes { `T = 1/X` } et { `X !num, X # 0, T = 1/X` } sont eux aussi équivalents.

Examinons d'autres exemples de systèmes de contraintes du même genre :

```
{ T = X[Y] }          (1)
{ T = <>[y] }         (2)
{ T = (~X)[Y] }       (3)
{ T = (+X)[Y] }       (4)
```

---

1 Comme en mathématiques, nous dirons que deux systèmes sont équivalents lorsqu'ils ont le même ensemble de solutions.

© PrologIA

Le terme `X[Y]` représente l'ensemble des arbres qui ont `X` pour étiquette et `Y` pour suite de fils c'est-à-dire, puisqu'aucune autre contrainte ne pèse sur `X` ou sur `Y`, l'ensemble de tous les arbres : la contrainte *(1)* est satisfaite quelque soit l'arbre donné pour valeur à `T`.

Les contraintes *(2)*, *(3)* et *(4)* n'ajoutent aucune autre limitation sur la suite des fils de `T` ; en revanche, faisant intervenir des opérateurs arithmétiques ou booléens, elles déterminent le type de l'étiquette de cet arbre. En définitive, ces quatre systèmes de contraintes sont respectivement équivalents à

```
{ }                                                    ( 1 )
{ T  !tuple }                                          ( 2 )
{ T  !boolt }                                          ( 3 )
{ T  !numt }                                           ( 4 )
```

Restons dans les exemples un peu “bizarres” et examinons les deux contraintes

```
{ +X  #  ~Y }                                          ( 1 )
{ +X  =  ~Y }                                          ( 2 )
```

Le système *(1)* est équivalent à { `X !num`, `Y !bool` }. En effet, les opérateurs présents imposent ces types aux variables `X` et `Y` et la relation `#` n'ajoute rien puisque pour tout nombre *n* et tout booléen *b* on a *n* `#` *b*. Pour la même raison, le système *(2)* est insoluble.

Signalons pour terminer que les expressions

```
{ +X  =  ~X }                                          ( 1 )
{ +X  #  ~X }                                          ( 2 )
```

seront rejetées par l'analyseur de Prolog III. Bien que d'un point de vue strictement syntaxique elles soient correctes, comme il n'existe aucun arbre qui soit simultanément numérique et booléen, il n'existe donc aucune affectation qui transforme ces contraintes en des conditions vérifiées sur des arbres.

9. Règles et requêtes

Syntaxe

Nous en arrivons enfin aux programmes Prolog III. Ceux-ci sont constitués de *règles* ; chacune comporte une tête, qui est un terme, et une queue formée d'une suite de buts et d'un système de contraintes. La suite de buts peut être vide, le système de contraintes aussi. Quand le système de contraintes est vide, il peut être omis.

<règle>
::= <tête>
-> { <but> } [**,** <système de contraintes>] **;**

<requête>
::= { <but> } [**,** <système de contraintes>] **;**

<but>
::= **/**
::= <terme>

<tête>
::= <terme>

Voici des exemples de règles correctes :

```
repas(X,Y,Z) -> hors_d_oeuvre(X) plat(Y) dessert(Z);

somme(<X>.L,S) -> somme(L,S0) , { S = S0 + X };

poules_lapins(P, L, P + L, 2P + 4L) -> ;
```

Pour activer un programme on doit composer une requête. Une *requête* est la même chose qu'une queue de règle. Voici des exemples de requêtes correctes :

© PrologIA

```
hors_d_oeuvre(X) plat(Y) dessert(Z);

repas(H, sole, D) , { H # salade };

poules_lapins(P, L, 16, 44);

circuit(A,B,C) / , { C|A&~B = 0', D=>C, D#B };
```

Signification d'un programme Prolog III

Examinons maintenant la signification d'un programme Prolog III. Elle est double : d'une part il constitue la définition récursive d'un ensemble d'arbres qui sont considérés comme des *faits vrais* par le programmeur ; c'est ce qu'on appelle la *sémantique déclarative* du langage. D'autre part, un programme se traduit, pour l'interprète de Prolog III, en une suite d'actions à exécuter, de deux sortes : des résolutions de systèmes de contraintes et des appels de procédures. Nous appellerons cela la *sémantique opérationnelle* de Prolog III.

L'ensemble des faits définis par un programme Prolog III

Un programme est fait de règles de la forme :

$$t_0 \rightarrow t_1\ t_2 \ldots t_n\ ,\ S\ ; \qquad (1)$$

où t_0 est un terme, $t_1 \ldots t_n$ sont des termes[1] et S est un ensemble de contraintes. Nous savons que chaque affectation A de l'ensemble des variables de cette règle transforme les termes t_i en des arbres $a_i = t_{i/A}$ et l'ensemble S de contraintes en un ensemble $C = S_{/A}$ de conditions sur des arbres. Par conséquent, une règle telle que *(1)* engendre tout un ensemble de règles portant sur des arbres, *une pour chaque affectation A qui rend vraies les conditions* $C = S_{/A}$, de la forme :

$$a_0 \rightarrow a_1\ a_2 \ldots a_n\ ; \qquad (2)$$

1 Nous ignorerons pour le moment le cas de la coupure /, qui peut figurer dans une queue de règle à la place d'un terme.

Une telle règle sans variable exprime une propriété logique, qui s'énonce : « si a_1, a_2 … a_n sont des faits, alors a_0 est un fait ».

Dans le cas $n = 0$ on a une règle sans queue :

$$a_0 \rightarrow ; \qquad (3)$$

la propriété logique correspondante est tout simplement « a_0 est un fait ».

Dans ces conditions, l'ensemble qu'un programme Prolog III définit est le plus petit ensemble de faits satisfaisant à toutes les propriétés logiques engendrées par les règles du programme. Ce dernier est donc composé de règles qui énoncent des faits explicites (les règles sans queue et sans variable, du type *(3)*) et de toutes les autres règles, qu'on peut voir comme des procédés de déduction pour produire les faits implicites.

Considérons par exemple le programme très simple :

```
prix(fleurs,120) ->;
prix(chocolats,90) ->;
prix(vin,50) ->;
prix(poisson_seche,40) ->;

cadeau(chocolats) ->;
cadeau(vin) ->;
cadeau(fleurs) ->;

cadeau_pas_cher(X) ->
      cadeau(X)
      prix(X,P) ,
      { P <= 100 };
```

Compte tenu des contraintes, la dernière règle donne lieu aux règles sur des arbres :

```
cadeau_pas_cher(poisson_seche) ->
      cadeau(poisson_seche)
      prix(poisson_seche,40);
```


© PrologIA

```
cadeau_pas_cher(chocolats) ->
      cadeau(chocolats)
      prix(chocolats,90);
cadeau_pas_cher(vin) ->
      cadeau(vin)
      prix(vin,50);
```

et puisque `cadeau(chocolats)`, `prix(chocolats, 90)`, `cadeau(vin)` et `prix(vin, 50)` sont des faits, on en déduira que `cadeau_pas_cher(chocolats)` et `cadeau_pas_cher(vin)` sont eux aussi des faits. Ce programme définit donc l'ensemble de faits :
{ `prix(fleurs, 120)`, `prix(chocolats, 90)`, `prix(vin, 50)`, `prix(poisson_seche, 40)`, `cadeau(chocolats)`, `cadeau(vin)`, `cadeau(fleurs)`, `cadeau_pas_cher(chocolats)`, `cadeau_pas_cher(vin)` }.
Seuls les deux derniers de ces faits n'étaient pas explicites.

Exécution d'un programme

Nous venons de montrer quelle est l'information implicite contenue dans un programme Prolog III. Examinons maintenant en quoi consiste son exécution par la machine. Cette exécution vise à résoudre le problème suivant : « étant donnés une suite $t_1\ t_2\ \ldots\ t_n$ de termes et un système S de contraintes, trouver les valeurs des variables qui transforment les termes t_i en des faits définis par le programme et les contraintes de S en des conditions verifiées »

Ce problème sera soumis à la machine sous la forme d'une requête

$$t_1\ t_2\ \ldots\ t_n\ ,\ S$$

Avec deux cas particuliers intéressants :

- si la suite $t_1\ t_2\ \ldots\ t_n$ est vide, alors la requête se résume à demander la résolution du système S

- si le système S est vide et $n = 1$ alors la requête se résume à la question « pour quelles valeurs des variables le terme t_1 se transforme en un fait défini par le programme ? »

Pour expliquer comment Prolog III calcule la réponse à ces questions nous allons introduire une machine abstraite. Il s'agit d'une machine non déterministe, dont l'unique instruction de base est décrite par les trois formules suivantes :

(1) $(W, t_0\ t_1\ \ldots\ t_n\ , S)$

(2) $s_0 \rightarrow s_1\ \ldots\ s_m\ , R$

(3) $(W, s_1\ \ldots\ s_m\ t_1\ \ldots\ t_n\ , S \cup R \cup \{ s_0 = t_0 \})$

La formule *(1)* représente l'état de la machine à un instant donné. La requête courante est $t_0\ t_1\ \ldots\ t_n\ , S$. W est l'ensemble des variables auxquelles on s'intéresse.

La formule *(2)* représente la règle du programme utilisée pour changer d'état. Au besoin, on a renommé certaines de ses variables, pour qu'elle n'en ait aucune de commune avec *(1)*. Les autres règles du programme auraient pu être employées aussi bien ; utiliser celle-ci plutôt qu'une autre résulte d'un choix.

La formule *(3)* représente le nouvel état de la machine, après l'application de la règle *(2)*. Le passage dans cet état n'est possible que si le système de contraintes $S \cup R \cup \{ s_0 = t_0 \}$ possède une solution pour laquelle chacun des termes de la suite $s_1\ \ldots\ s_m\ t_1\ \ldots\ t_n$ représente un arbre bien défini.

© PrologIA

Pour répondre à la requête $t_0 \ldots t_n$, S la machine partira de l'état initial (W, $t_0 \ldots t_n$, S), où W est l'ensemble des variables qui figurent dans la requête, et passera dans tous les états qu'elle peut atteindre en répétant l'opération de base ci-dessus. Bien entendu, le choix exprimé par la formule *(2)* n'est pas laissé au hasard : Prolog III fait en sorte que toutes les règles soient successivement essayées[1], dans l'ordre où elles ont été écrites dans le programme. Ainsi, la machine Prolog III est en réalité la machine (déterministe) qui produit toutes les exécutions possibles de la machine non déterministe définie par les expressions *(1)*, *(2)*, *(3)*.

Chaque fois qu'il parviendra à un état de la forme

(W, , S)

c'est-à-dire dont la suite de buts est vide, Prolog III fournira comme réponse la solution du système S sur l'ensemble de variables W.

Par exemple, à partir de la requête très simple

```
cadeau_pas_cher(X);
```

des états de la machine seront (on ne montre pas les étapes promises à échouer) :

```
{X}, cadeau_pas_cher(X) , {};
```

utilisation de la règle `cadeau_pas_cher(X) ->` *etc…* :

```
{X}, cadeau(X')  prix(X',P') ,

  {P≤100,cadeau_pas_cher(X)=cadeau_pas_cher(X')};
```

qui se simplifie en :

```
{X}, cadeau(X')  prix(X',P') , {P≤100,X=X'};
```

---

1 Ou tout au moins toutes les règles *raisonnables*, c'est-à-dire toutes sauf celles qui de manière évidente rendraient le système $S \cup R \cup \{ s_0 = t_0 \}$ insoluble.

utilisation de la règle `cadeau(chocolats) ->;` :

```
{X}, prix(X',P')  ,

  {P'≤100,X=X',cadeau(X')=cadeau(chocolats)};
```

qui se simplifie à son tour en :

```
{X}, prix(X',P')  ,
          {P'≤100,X=X',X'=chocolats};
```

utilisation de la règle `prix(chocolats,90) ->;` :

```
{X}, ,    {P'≤100,X=X',X'=chocolats,
                  prix(X',P')=prix(chocolats,90)};
```

lequel se simplifie en :

```
{X},  ,    {P'≤100,X=X',X'=chocolats,P'=90)};
```

La suite de buts étant devenue vide, Prolog III affiche alors la solution du système courant *S* = `{P'≤100,X=X',X'=chocolats,P'=90)}`, ou plutôt la partie intéressante de cette solution, soit

```
{X = chocolats}
```

ensuite Prolog III explore les autres exécutions possibles de la machine non déterministe *(1), (2), (3)* ; c'est ainsi que la solution

```
{X = vin}
```

est affichée peu après.

ststs

© PrologIA

Association Prolog
HERITAGE

Arbres, tuples, chaînes et listes

1. **Introduction**
2. **Les arbres**
3. **Les tuples**
4. **Règles prédéfinies sur les tuples**
5. **Les listes**
6. **Les chaînes**
7. **Exemples**

Que trouve-t-on dans ce chapitre ?

Le domaine de Prolog III est celui des arbres (finis et infinis). Un nouvel opérateur universel a été ajouté par rapport aux Prolog standard : le constructeur général d'arbres. La notion de liste a également été étendue à celle de tuple, et ce domaine muni d'une véritable opération de concaténation. Les chaînes de caractères sont, d'autre part, un sous-ensemble des tuples. Enfin, pour des raisons de compatibilité et d'efficacité, les listes classiques de Prolog ont été conservées. Ce chapitre présente les opérations et les relations définies sur les arbres et les tuples, ainsi qu'un certain nombre d'exemples utilisant des contraintes sur ces domaines.

© PrologIA

1. Introduction

Prolog III améliore sensiblement le traitement des arbres, en introduisant un opérateur qui autorise la représentation d'un arbre de manière totalement générique, indépendamment de son arité. En ce qui concerne les listes, structures de données de base des programmes Prolog, l'amélioration apportée par Prolog III est fondamentale et introduit un nouvel objet : le tuple. Cette évolution répond à un double constat. Tout d'abord, la structure même de liste, un arbre construit à partir d'un symbole fonctionnel binaire, rend l'opération de concaténation très onéreuse. Ensuite, tout accès au n-ième élément d'une liste (y compris lors d'une concaténation programmée en Prolog) ne peut être effectué qu'au travers d'un parcours séquentiel. Ces deux inconvénients sont supprimés grâce à l'introduction de contraintes sur les tuples, munis d'une concaténation dont les propriétés répondent à ce que l'on est en droit d'attendre.

2. Les arbres

Les arbres forment la classe la plus générale d'objets définis en Prolog III. Tous les objets définis dans la suite de ce document (tuples, listes, chaînes, valeurs numériques ou booléennes, …) sont des éléments de l'ensemble des arbres et toute variable Prolog III représente une valeur prise dans cet ensemble.

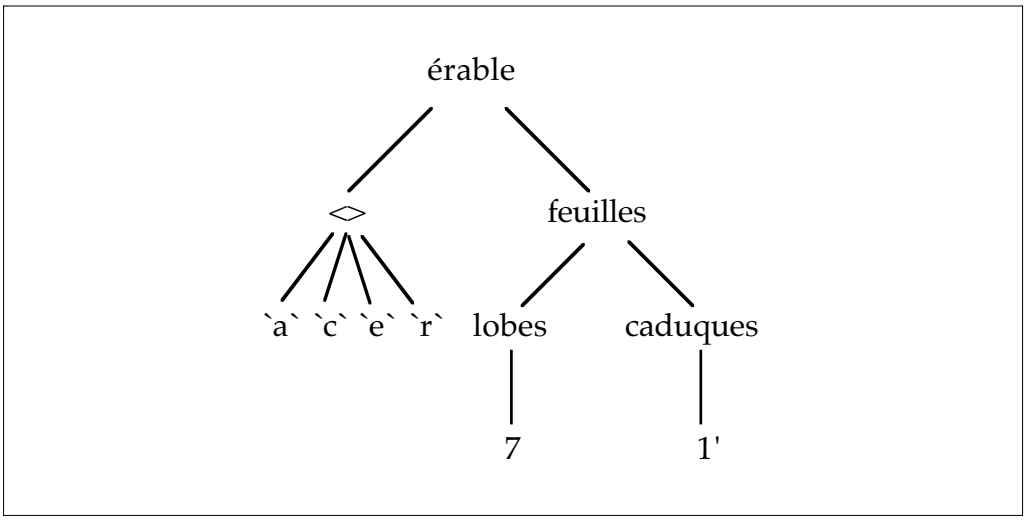
Ces arbres sont composés de nœuds étiquetés par :

- des identificateurs,
- des caractères,
- les valeurs booléennes `0'` et `1'`,
- des valeurs numériques ,
- le signe spécial `<>`

{ On prendra bien garde à ne pas confondre arbres et termes. Les arbres sont des éléments du domaine de Prolog III ; les termes sont des constructions syntaxiques.

© PrologIA

Voici un exemple d'arbre :

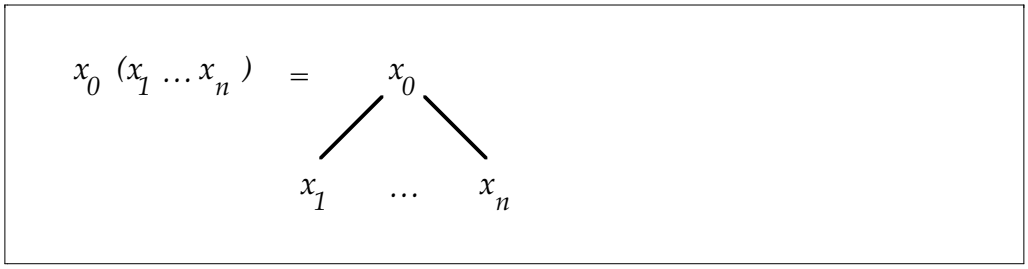


Les arbres dont l'étiquette initiale est un identificateur sont appelés *arbres factuels*, et ceux dont l'étiquette initiale est le signe `<>` sont appelés *tuples*. Les tuples dont tous les éléments sont des caractères sont appelés *chaînes*.

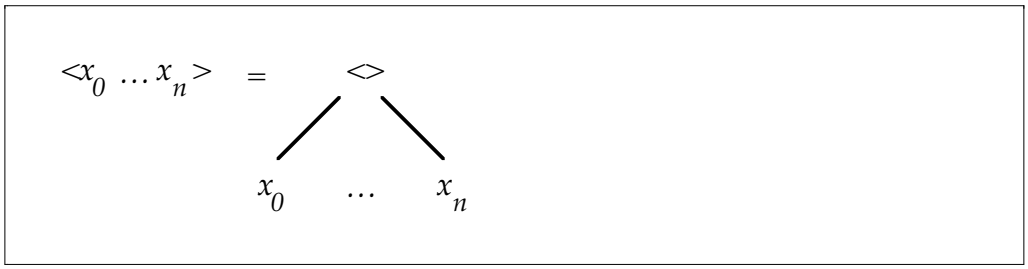
{ NOTATION : En Prolog III, les chaînes sont plutôt représentées entre guillemets (par exemple : `"Ralliez vous à mon panache blanc"`) et les tuples entre chevrons (par exemple `<Marignan, 1515, 1'>`).

Opérations sur les arbres

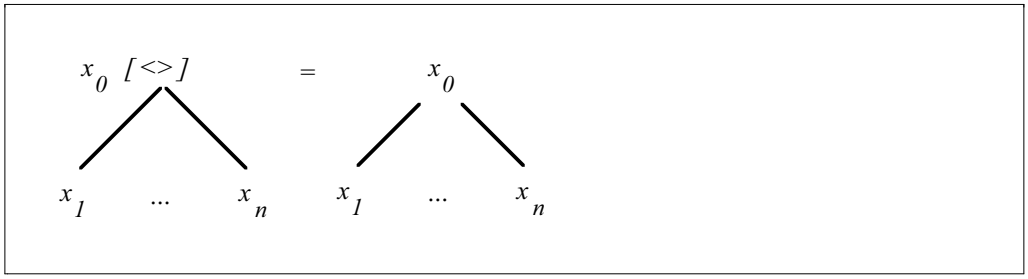
Les opérations définies sur les arbres sont les opérations de construction. On trouve ainsi, comme en Prolog II le *constructeur d'arbres,* mais également deux nouveaux opérateurs, le *constructeur de tuple,* et le *constructeur général d'arbres,* qui permet de désigner tout arbre de manière totalement générique, indépendamment de son nombre de fils. Voici les schémas illustrant ces opérations.



Le constructeur d'arbres



Le constructeur de tuples



Le constructeur général d'arbres

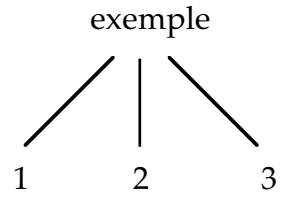
Le constructeur général d'arbres permet de représenter tout arbre à l'aide seulement de son étiquette initiale et du tuple formé de ses fils immédiats.

{ ATTENTION : on ne peut pas insérer d'espaces entre le terme qui représente l'étiquette de l'arbre et la parenthèse ouvrante du constructeur. C'est là l'un des rares cas en Prolog III où l'écriture d'un blanc est interdite.

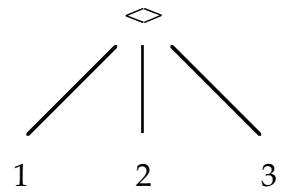
Voici quelques exemples de l'utilisation de ces divers constructeurs. On donne ici plusieurs formes syntaxiques possibles qui utilisent les différents constructeurs, et les arbres correspondants :

© PrologIA

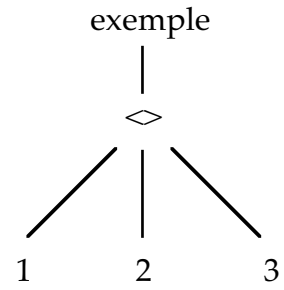
```
exemple(1,2,3)
exemple[<1,2,3>]
```




```
<1,2,3>
<>(1,2,3)
<>[<1,2,3>]
```

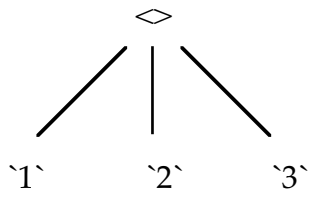



```
exemple(<1,2,3>)
exemple[<<1,2,3>>]
```



Voici enfin un exemple du même type portant sur une chaîne de caractères :

```
"123"
<>(`1`,`2`,`3`)
<>["123"]
<>[<`1`,`2`,`3`>]
```



Contraintes sur les arbres

Les contraintes sont construites à l'aide de relations. Les relations utilisées sur les arbres sont :

- l'égalité (`=`),
- l'inégalité (`#`),
- la relation unaire qui impose à un arbre d'avoir un nombre déterminé de fils immédiats. Dans le cas où cet arbre est représenté par la variable `A` et son nombre de fils par `N`, cette relation se note `A::N.`

Exemples

Voici quelques exemples de contraintes sur les arbres.

```
{X  =  Erable("acer",feuilles(lobes(7),caduques(1')))}
```

Cette contrainte impose à la variable `X` de représenter l'arbre donné en exemple précédemment.

```
{X = E[U],Y = E[V]}
```

Cet ensemble de contraintes impose aux variables `X` et `Y` de représenter deux arbres dont les étiquettes initiales sont égales

```
{X = E[U], E # <>}
```

Cette contrainte impose à la variable `X` de représenter un arbre qui ne soit pas un tuple.

Examinons enfin quelques exécutions qui illustrent l'utilisation de ce type de contraintes :

© PrologIA

```
> {E[U]  =  exemple(1,2,3)};
  {E = exemple, U = <1,2,3>}

> {E[U]  =  <1,2,3>};
  {E = <>, U = <1,2,3>}

> {E[U]  =  exemple(<1,2,3>)};
  {E = exemple, U = <<1,2,3>>}

> {E[U]  =  "123"};
  {E = <>, U = "123"}

> {E[U]  =  121/3};
  {E = 121/3, U = <>}
```

Les exemples suivants montrent comment sont simplifiées certaines équations sur les arbres :

```
> {X(Y)  =  Y(X)  };
  {Y = X, X::0 }

> {<X>  =  X[Y]};
  {X = <>, Y = <<>>}

> {Y[X]  =  X[Y]  };
  {Y = <>, X = <>}
```

Voici enfin des exemples qui montrent des contraintes pour lesquelles les solutions sont des arbres infinis :

```
> {X(Y)  =  X[Y]};
  {Y = <Y>, X::0 }

> {X(Y)  =  <<X,Y>>};
  {X = <>, Y = <<>,Y>}
```

Restrictions sur la contrainte de taille

En Prolog III on ne peut imposer à un arbre d'avoir un nombre déterminé de fils immédiats qu'à la condition que ce nombre soit une constante, c'est-à-dire un entier positif. Toutefois, pour augmenter la facilité de programmation, si ce nombre est représenté par un terme contenant des variables un mécanisme de retardement est mis en place par Prolog III. Nous reviendrons sur ce mécanisme dans le chapitre spécialement consacré aux retardements.

```
{E[<X, ff(Y)>. Z] :: 13};
```

Dans une contrainte de taille, le nombre de fils immédiats doit être explicitement connu

3. Les tuples

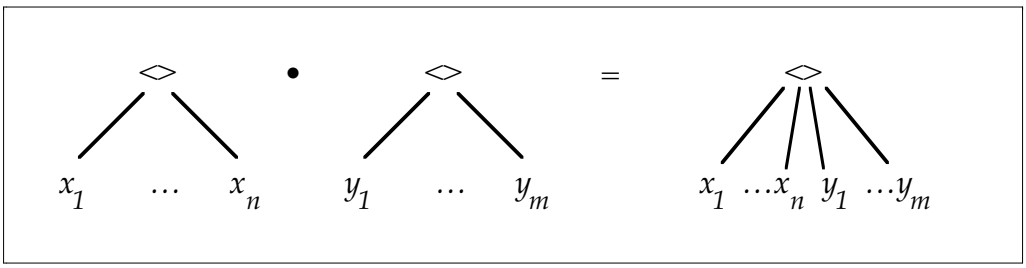
On appelle *tuple* tout arbre dont l'étiquette initiale est <>. Cette première définition permet de considérer les tuples comme des suites finies d'arbres. On notera au passage que l'aspect fini se rapporte au nombre d'éléments (et non pas à la profondeur de ceux-ci), et que le caractère de suite induit un ordre sur ces éléments (les tuples ne sont pas des ensembles). On appellera souvent *taille* le nombre de fils d'un tuple.

Opérations

La seule opération définie sur les tuples est l'opération de *concaténation*, notée par un point (“`.`”). Cette opération construit, à partir de deux tuples `T1` et `T2`, respectivement de tailles `n` et `m`, le tuple `T1.T2`, de taille `m+n`, dont les `n` premiers fils sont ceux de `T1`, et les `m` suivants ceux de `T2`.

© PrologIA

Ceci peut s'illustrer par le schéma suivant :



Concaténation de deux tuples

Relations

Les relations qui permettent de construire des contraintes sur les tuples sont les mêmes que celles utilisées pour les arbres, à savoir l'égalité, l'inégalité, et la relation unaire permettant d'associer à un tuple le nombre de ses éléments. Dans ce dernier cas, on dispose de la relation de taille sur les arbres, mais également d'une relation unaire qui impose à un terme `T` de représenter un tuple (`T !tuple`).

Restrictions concernant les tuples

Pour réaliser un bon compromis entre la complexité des contraintes traitées et les considérations d'efficacité liées à la réalisation d'un langage de programmation, Prolog III impose un certain nombre de restrictions sur ces contraintes. En ce qui concerne les tuples, cette restriction porte principalement sur le fait que *tout tuple figurant en membre gauche d'une opération de concaténation doit être de taille explicitement connue.* La seconde restriction découle naturellement de celle imposée aux arbres, à savoir que pour toute contrainte de taille portant sur un tuple, le nombre représentant celle-ci doit être explicitement connu. Dans le cas où ces restrictions ne sont pas respectées, un mécanisme de retardement, que nous détaillerons par la suite, est mis en place (voir le chapitre entièrement consacré aux retardements)

```
{   U.V.W = <E>.Z,
    U :: 4, V :: 2   }
```

Tout tuple figurant en membre gauche d'une opération de concaténation doit être de taille explicitement connue

Exemples de contraintes sur les tuples

Voici un certain nombre de systèmes de contraintes corrects portant sur des tuples :

```
{<1,2,3>.V  =  Z}

{<E>.U  =  V}

{U.V  =  W,U  ::  100}

{U.<E,F>.V  =  W,  U  ::  4}

{(U.V).W  =  T,U  ::  2,V  ::  4}
```

On notera dans ces exemples que pour toute concaténation, la taille des opérandes gauches est connue.

Voici, cette fois quelques exécutions de ce types de contraintes, qui montrent l'ensemble des solutions correspondantes :

```
> {<0>.U  =  U.<0>,  U  ::  10};
  {U = <0,0,0,0,0,0,0,0,0,0>}

> {Z  ::  10,  <1,2,3>.Z  =  Z.<2,3,1>};
  {Z = <1,2,3,1,2,3,1,2,3,1>}
```


© PrologIA

Récapitulatif des opérations et relations

Pour récapituler, voici toutes les opérations et relations utilisables sur les arbres et les tuples :

Opérations et relations définies sur les arbres

opérations

constructeur d'arbre	$x_0(x_1, \ldots, x_n)$
constructeur général d'arbre	$x_0[<x_1, \ldots, x_n>]$

relations

égalité	`x = y`
inégalité	`x # y`
taille	`x :: n`

Opérations et relations définies sur les tuples

opérations

constructeur de tuple	$<x_0, x_1, \ldots, x_n>$
concaténation	`u.v`

relations

égalité	`u = v`
inégalité	`u # v`
taille	`u :: n`
typage	`u !tuple`

4. Règles prédéfinies sur les tuples

Les règles prédéfinies qui concernent la taille des tuples se divisent, on aura l'occasion de le revoir souvent, en deux classes : les règles occasionnant un retardement et celles pour lesquelles ce n'est pas le cas. On retrouvera une description détaillée et des exemples dans le chapitre "Règles prédéfinies et procédures externes".

Tailles et concaténations

bound_size(T,N) : Pose la contrainte {T :: N} lorsque N est un entier positif connu, ou {N = k} si k est la taille de T. Echoue si T n'est pas de taille connue et si N est inconnu ou ne représente pas une valeur entière positive.

size(U,N) : Pose la contrainte {U :: N}, avec les retardements nécessaires si N et la taille de U ne sont pas connus. Echoue si N est connu mais ne représente pas une valeur entière positive.

bound_conc(U1,U2,U3) : Pose la contrainte {U3 = U1.U2}. Echoue si la taille du tuple U1 n'est pas connue.

conc3(U1,U2,U3) : Pose l'ensemble de contraintes suivant : { U3 = U1.U2, U1 :: N1, U2 :: N2, U3 :: N3, N3 = N1+N2, N1 >= 0, N2 >= 0}, avec les éventuels retardements lorsque N1, N2 *ou* N3 ne sont pas connus.

arg3(N,T1,T2) : Cette primitive pose la contrainte {T2 = N'}, où N' est le *Nième* argument du tuple T1. Si N est nul, T2 est égal à l'étiquette initiale de T1 c'est-à-dire <>. Echoue si N n'est pas un entier positif connu ou si T1 n'est pas un tuple.

© PrologIA

Divers

arg(N,T1,T2) : Cette primitive pose la contrainte {T2 = N'}, où N' est le *Nième* argument du terme T1. Si N est nul, T2 est égal à l'étiquette initiale de T1. En particulier, si T1 est un tuple T2 représente le *Nième* élément de ce tuple. Echoue si N n'est pas un entier positif connu.

known_part(U1,U2,U3) : Pose la contrainte {U1 = U2.U3}, où U2 est un tuple formé de la plus grande suite de taille connue des premiers éléments de U1.

split(U,L) : Pose la contrainte {L = L'}, où L' est la liste composée des éléments qui forment le tuple U (voir la suite de ce chapitre concernant les listes). Echoue lorsque le tuple U n'est pas connu à l'exécution.

tuple(U) : S'exécute avec succès si U représente un tuple entièrement connu (voir le prédicat *bound*). Echoue dans tous les autres cas.

is_tuple(U) : Vérifie que le terme U représente un tuple c'est-à-dire que la contrainte {U !tuple} appartient au système courant. Echoue dans le cas contraire.

Un exemple de programme sur les tuples

Comme premier exemple de programme, nous nous proposons de considérer le retournement (ou inversion) d'un tuple. Nous examinerons successivement la manière habituellement utilisée en Prolog, puis la programmation en Prolog III, et nous terminerons par l'examen d'un programme utilisant des contraintes retardées.

Voici tout d'abord le programme classique, à ceci près que l'on utilise des tuples au lieu de listes. On pourra en profiter pour noter la manière d'exprimer un tuple à la mode “Prolog standard” (un couple tête-de-liste, queue-de-liste) à l'aide de la concaténation. Le rapport, dans ce cas, entre le E.L de Prolog II et le <E>.L de Prolog III n'est, on l'aura compris que purement fonctionnel, la sémantique du . (point) demeurant fondamen-

talement différente. On retrouvera également dans ce programme, peut être avec une certaine émotion, le toujours fameux prédicat **conc.**

```
naive_reverse(<>,<>) -> ;
naive_reverse(<E>.X,X') ->
  naive_reverse(X,X'')
  conc(X'',<E>,X') ;

conc(<>,X,X) -> ;
conc(<E>.X,Y,<E>.Z) ->
  conc(X,Y,Z) ;
```

Voici maintenant le même programme, qui utilise cette fois-ci la concaténation de Prolog III. Le prédicat **bound_conc** installe la contrainte **{X' = X''.<E>}** après l'exécution de **naive_reverse**. On est alors certain de connaître la taille du tuple **X''.** Il n'y a donc aucun retard mis en place. Cette manière de faire est bien plus efficace que la première, dans la mesure où la concaténation de Prolog III est bien supérieure au prédicat **conc** décrit dans le programme précédent.

```
naive_reverse(<>, <>) -> ;
naive_reverse(<E>.X,X') ->
  naive_reverse(X,X'')
  bound_conc(X'',<E>,X') ;
```

Voici enfin un troisième programme, qui met en place les concaténations avant l'exécution de naive_reverse. Des contraintes retardées sont alors mises en place (voir chapitre sur les retardements). Si le programme fournit les résultats attendus, les temps d'exécutions sont moins bons que dans le cas précédent, en raison de la gestion des retards.

```
naive_reverse(<>,<>) -> ;
naive_reverse(<E>.X,X'.<E>) ->
  naive_reverse(X,X') ;
```


© PrologIA

5. Les listes

Pour des raisons d'efficacité, et bien que l'opérateur binaire utilisé pour les listes en Prolog classique puisse être aisément remplacé par une concaténation, Prolog III permet l'utilisation de listes “classiques” construites sur la notion de paire pointée, à l'aide d'un symbole fonctionnel binaire représenté par `[]`.

Syntaxiquement les formes suivantes sont permises :

`[U|L]`	qui représente une liste de tête `U` et de queue `L`
`[U1,U2,...,Un]`	qui représente une liste formée des éléments `U1,U2,...,Un`
`[U1,U2,...,Un|L]`	qui représente une liste dont les premiers éléments sont `U1,U2,...,Un` et la queue de liste `L`

{ Bien que la syntaxe ne soit pas ambiguë, vous devez veiller à ne pas confondre l'opérateur binaire utilisé pour les listes et représenté par des crochets (par exemple `[U]`, pour une liste réduite à un élément), et l'opérateur n-aire de construction générale d'arbres, représenté par le même symbole (par exemple `E[U]`).

{ ATTENTION : il y a une véritable ambiguïté dans le fait suivant : comment distinguer une liste ayant `A` pour tête et `B` pour queue, qu'on doit noter `[A|B]`, d'une liste réduite à l'unique booléen, résultat de la disjonction des booléens `A` et `B`, qu'on devrait noter encore `[A|B]` ? Cette ambiguïté est levée en imposant conventionnellement, dans le second cas, le parenthésage de l'expression booléenne en question : `[(A|B)]`

Il n'y a aucune opération définie sur les listes en Prolog III autre que celles concernant les arbres, et donc pas de contraintes particulières. On pourra quelquefois privilégier ce type de structures dans des programmes sans concaténations pour en améliorer les performances.

{ On peut se demander à juste titre quel est le statut exact du double symbole `[]`, qui désigne la liste vide. La réponse est qu'il s'agit d'un identificateur, qui devrait en toute rigueur être représenté entre quotes, mais pour lequel il est toléré d'omettre ces quotes. Les notations `[]` et `'[]'` (et même `sys:[]`) sont donc équivalentes.

Voici quelques exécutions faisant intervenir des listes :

```
> {E[U]  =  [A,B,C|L]};
  {E = '[]', U = <A,[B,C | L]>}

> {E(X,Y)  =  [A,B,C|L]};
  {E = '[]', Y = [B,C | L], A = X}

> {[A,B,C|L]  =  [A'|L']};
  {A' = A, L' = [B,C | L]}

> is_ident([]);
  {}
```

Primitives sur les listes

`arg2(N,L,T)` : Si `N` est nul, pose la contrainte `{T = N'}`, où `N'` représente le nombre d'élément de la liste `L`. Si `N` n'est pas nul, pose la contrainte `{T = N'}`, où `N'` est le *Nième* argument de la liste `L`. Echoue si `N` n'est pas un entier positif connu ou si la liste `L` n'est pas suffisamment connue.

`list_tuple(L,U)` : Pose la contrainte `{U = U'}`, où `U'` est le tuple composé des éléments qui forment la liste `L` (voir la suite de ce chapitre concernant les listes). Echoue lorsque la liste `L` n'est pas entièrement connue.

6. Les chaînes

Comme nous l'avons déjà mentionné, les chaînes en Prolog III sont des tuples dont les éléments sont des caractères. On peut donc utiliser sur les

© PrologIA

chaînes l'opération de concaténation ainsi que les relations unaires et binaires utilisables sur les tuples.

Voici quelques exemples de contraintes sur des chaînes :

```
> {X."a" = "a".X, X :: 10};
  {X = "aaaaaaaaaa"}

> {X = Y." une concatenation ".<`d`,`e`,` `>.Z,
  Y = <>(`V`,`o`,`i`,`c`,`i`), Z = U["chaine"]};
  {X = "Voici une concatenation de chaine",
  Y = "Voici",
  Z = "chaine", U = <>}
```

Primitives sur les chaînes

Il nous faut également noter un certain nombre de règles prédéfinies et prédicats évaluables concernant les chaînes de caractères. On se reportera pour plus de détails au chapitre *Règles prédéfinies et procédures externes.*

Manipulations de chaînes

`conc_string(S1,S2,S3)` : Cette règle prédéfinie a le même effet que son homologue de Prolog II. Elle énumère tous les triplets `S1,S2,S3` de chaînes telles que `S3` est la concaténation de `S1` et `S2`. Ce prédicat ne réussit que si les chaînes `S1` et `S2` ou bien la chaîne `S3` sont entièrement connues.

`find_pattern(S1,S2,N)` : Pose la contrainte `{N = D}` où *`D`* est la position du début de la chaîne `S2` dans la chaîne `S1`. Si la chaîne `S2` n'est pas trouvée, ou si `S1` et `S2` ne sont pas entièrement connues, alors `find_pattern` échoue.

`substring(S1,N1,N2,S2)` : Pose la contrainte `{S2 = S1'}`, où `S1'` est la sous-chaîne de `S1` commençant à la position `N1`, de longueur `N2`. Echoue si une telle sous-chaîne n'existe pas ou si `S1` n'est pas connue.

Conversions de chaînes

list_string(L,S) : Pose la contrainte {S = L'}, où L' est une chaîne composée des caractères qui forment la liste L. Echoue si la liste L n'est pas entièrement connue.

string_ident(P,S,I) :

- Si I est connu : pose l'ensemble de contraintes {S = S',P = P'} où S' est une chaîne composée des caractères de la représentation abrégée de l'identificateur I, et P' une chaîne composée des caractères formant le préfixe de l'identificateur I.
- Si S et P sont connus : pose l'ensemble de contraintes {I = I'}, où I' est un identificateur composé des caractères de la chaîne représentant son préfixe, P, et de celle représentant sa notation abrégée, S.

string_integer(S,N) :

- Si S est connu : pose l'ensemble de contraintes {N = S'} où S' est un entier formé à partir des caractères de la chaîne S.
- Si N est connu : pose l'ensemble de contraintes {S = N'}, où N' est une chaîne formée des caractères qui composent l'entier N.

string_real(S,F) :

- Si S est connu : pose l'ensemble de contraintes {F = S'} où S' est un flottant formé à partir des caractères de la chaîne S.
- Si F est connu : pose l'ensemble de contraintes {S = F'}, où F' est une chaîne formée des caractères qui composent le flottant F.

string_bool(S,B) :

- Si S est connu : pose l'ensemble de contraintes {B = S'} où S' est un booléen formé à partir des caractères de la chaîne S.
- Si B est connu : pose l'ensemble de contraintes {S = B'}, où B' est une chaîne formée des caractères qui composent le booléen B.

Divers

string(S) : S'exécute avec succès si S représente une chaîne entièrement connue. Echoue dans tous les autres cas.

© PrologIA

7. Exemples

Nous proposons ici un certain nombre de programmes, concernant plus particulièrement les tuples, qui montrent des possibilités d'utilisation. Certains utilisant des contraintes retardées, il serait bon de se reporter au chapitre correspondant si l'on désire les examiner dans le détail.

Calcul des feuilles d'un arbre

Le programme suivant recherche les feuilles différentes de <> dans un arbre. On notera la puissance de l'opérateur général de construction d'arbres, ainsi que l'utilisation des contraintes retardées.

```
feuilles(<>,<>) ->;
feuilles(A[<>], <A>) ->, {A#<>};
feuilles(A[<U>.V], X.Y) ->
  feuilles (U, X)
  feuilles (V, Y);
```

Voici une exécution de ce programme :

```
> feuilles(Erable("acer",feuilles(lobes(7),
              caduques(1'))), L);
  {L = "acer".<7,1'>}
```

Quick sort

Voici un programme de tri récursif, le "quick sort", qui utilise les contraintes sur les tuples. Rappelons brièvement l'algorithme utilisé. Pour trier un tuple composé de nombres, on choisit un élément, on partitionne le tuple en deux, le premier tuple contenant les éléments inférieurs ou égaux à l'élément considéré, le second les éléments strictement supérieurs. On applique alors récursivement l'algorithme sur les deux tuples ainsi formés jusqu'à arriver au tuple vide.

Le choix de l'élément privilégié peut s'effectuer de diverses manières. Nous avons choisi ici, afin de mettre en évidence l'accès direct, de choisir l'élément situé au milieu du tuple.

```
trier(<>,<>) -> ;
trier(L,U.<X>.V) ->
  partitionner(L,X,U',V')
  trier(U',U)
  trier(V',V);

partitionner(L,X,U',V') ->
  trunc(N/2,N')
  regrouper(X,L',U',V'),
  { L :: N,
    U :: N',
    L = U.<X>.V,
    L' = U.V};

regrouper(X,<>,<>,<>) ->;
regrouper(X,<X'>.L,<X'>.U,V) ->
  regrouper(X,L,U,V),
  { X' <= X};
regrouper(X,<X'>.L,U,<X'>.V) ->
  regrouper(X,L,U,V),
  { X' > X};
```

On peut proposer un certain nombre d'exécutions de ce programme. la première consiste à effectuer un tri sur un tuple entièrement connu :

```
> trier(<6,5,4,3,2,1>,L>);
  {L = <1,2,3,4,5,6>}
```

On peut également chercher les tuples qui, une fois triés, donnent un tuple connu. Par exemple :

© PrologIA

```
> trier(L,<1,2,3>) { L :: 3 };
  {L = <1,2,3>}
  {L = <1,3,2>}
  {L = <2,1,3>}
  {L = <2,3,1>}
  {L = <3,2,1>}
  {L = <3,1,2>}
>
```

Cette exécution fournit, bien sûr, toutes les permutations de *n* éléments.

Enfin, on peut trier un tuple dont les éléments sont inconnus, mais sur lesquels on connait déjà un ordre :

```
> trier(<X,Y,Z>,L),{Z>Y>=X};
  {Y = X + X1, Z = X + X2 + X1,
   L = <X,X + X1,X + X2 + X1>,
   X1 >= 0 ,
   X2 > 0 }
```

Les variables d'écart introduites ont été renommées **X1** et **X2.**

Une suite périodique

Le but de ce programme est de montrer que la suite définie par :

$$X_{i+2} = |X_{i+1}| - X_i$$

est périodique de période 9

Voici le programme :

```
Suite(<Y,X>) ->;
Suite(<Y'-X,Y,X>.U) ->
     Suite(<Y,X>.U)
     Valeur_absolue(Y,Y');
```

```
Valeur_absolue(Y,Y) -> , { Y >= 0 };
Valeur_absolue(Y,-Y) -> , { Y < 0 };
```

Ce programme se contente de décrire la construction d'un tuple dont les éléments constituent une suite d'éléments de la suite définie précédemment.

Pour prouver que cette suite est périodique de période 9, toute la subtilité consiste à poser une requête qui peut s'énoncer de la manière suivante :

Quels sont les tuples de 11 éléments décrits par ce programme qui sont tels que l'un ou l'autre des deux premiers éléments soient différents de l'un ou l'autre des deux derniers éléments. Voici cette requête :

```
> Suite(U.V.W) ,
  { U :: 2, V :: 7, W :: 2, U # W };
>
```

La requête échoue, ce qui signifie que l'ensemble des solutions est vide, et prouve donc bien la propriété (!!). Elégant, non ?

Crible d'Eratosthène

Ce programme, qui calcule les nombres premiers inférieurs à un entier `N`, peut s'écrire de manière plus efficace sans utiliser l'arithmétique sur la taille des tuples. Il constitue cependant un excellent exemple de maniement des tuples et de la concaténation. Voici le programme :

© PrologIA

```
nombres_premiers(<1'>.Z) ->
  cribles(2, X, Z),
  {Z :: N, X :: N, X.<1'> = <1'>.X};

cribles(N, <>, <>) ->;
cribles(N, <0'>.X, <0'>.Z) ->
  cribles(N+1, X, Z);
cribles(N, <1'>.X, <1'>.Z) ->
  crible(N, X, Y)
  cribles(N+1, Y, Z);

crible(N, X, X) -> {X :: M, N > M};
crible(N, X, Y) ->
  Crible(N, X', Y')
  { X :: M, N <= M,
    X = U.<E>.X',
    Y = U.<0'>.Y',
    U :: N-1 };
```

Pour construire le tuple formé de valeurs booléennes où le rang de tout élément égal à **1'** est un nombre premier, on procède de la manière suivante (prédicat **cribles**) : pour un élément donné, et un nombre donné **N** (en commençant à 2, et en initialisant le tuple de manière à ce que tous les éléments soient égaux à **1'**), si sa valeur est **0'**, on la renvoie, sinon on renvoie **1'**, et on unifie à **0'** tous les éléments multiples de **N** par simple décalage (i.e. à l'aide de la concaténation). Cette dernière opération est réalisée par le prédicat **crible.**

Voici une exécution :

```
> nombres_premiers(Y), {Y :: 12};
  {Y = <1',1',1',0',1',0',1',0',0',0',1',0'>}
```

ststs

Association Prolog
HERITAGE

Les contraintes numériques

1. **Introduction**
2. **Généralités**
   - **Les nombres**
   - **Expressions numériques**
   - **Les relations**
   - **Contraintes numériques**
   - **Exemples de contraintes numériques**
   - **Restrictions**
   - **Forme normale**
3. **Règles prédéfinies et procédures externes spécifiques**
4. **Retardement des contraintes non-linéaires**
5. **Formats d'entrée-sortie**
6. **Exemples de programmes**
   - **Calcul bancaire**
   - **Crypto-arihmétique**
   - **Remplissage d'un rectangle par des carrés**

Que trouve-t-on dans ce chapitre ?

Parmi les différents domaines sur lesquels on peut faire porter des contraintes en Prolog III, celui pour lequel on pressent d'ores et déjà le plus d'applications est certainement le domaine numérique. En effet, la possibilité, offerte en Prolog III, de résoudre des systèmes d'équations et d'inéquations sur les nombres rationnels et réels ouvre la voie de la programmation logique à tous les problèmes de type ordonnancement, planification, réseaux, analyse financière, etc. Nous aurons l'occasion dans ce chapitre de développer des exemples de ce type, non sans avoir au préalable exposé les diverses possibilités offertes par le langage et concernant ce type de contraintes, de la description syntaxique des systèmes numériques jusqu'à l'utilisation de certains prédicats évaluables spécifiques.

© PrologIA

1. Introduction

Prolog a toujours été considéré, à juste titre, comme un puissant langage symbolique, impropre à tout traitement numérique. Ceci est lié en grande partie à des considérations d'ordre historique, et bons nombres d'interpréteurs et de compilateurs du langage se sont jusqu'à présent bornés à ajouter des couches coiffant l'unification, et permettant de manipuler certaines expressions numériques. Ces traitements consistent en fait à évaluer de telles expression en effectuant des calculs sur des variables dont les valeurs sont connues au moment de l'unification. Prolog III propose une approche qui va bien au-delà de ce simple «sucre syntaxique», puisque les variables représentant des nombres sont traitées comme toutes les variables Prolog III, c'est à dire que ce sont des objets dont la valeur peut parfaitement être inconnue. Ainsi, les expressions numériques ne sont généralement pas évaluées (les valeurs de suffisamment de variables n'étant pas connues) mais forment les principaux composants des contraintes numériques traitées par le langage au niveau de l'unification. Ce traitement consiste principalement à vérifier si l'ajout de ces contraintes ne rend pas le système insoluble ainsi qu'à effectuer un certain nombre d'autres traitements sur lesquels nous reviendrons par la suite.

2. Généralités

Les nombres

Les contraintes numériques de Prolog III portent sur deux types d'objets : les nombres rationnels et les nombres flottants.

Les nombres rationnels sont représentés dans la machine en précision parfaite, c'est-à-dire avec autant de chiffres que leur expression exacte le requiert. En fait, tout rationnel est codé sous la forme d'un nombre fractionnaire irréductible, par un couple d'entiers en précision parfaite qui représentent son numérateur et son dénominateur. Ainsi Prolog III garantit l'unicité de la représentation de tout nombre rationnel (les nombres `3/9`, `2/6` et `1/3` représentent le même nombre rationnel et sont codés sous la

© PrologIA

forme `(1,3)`). Pour conserver cette unicité de la représentation, un module arithmétique en précision parfaite a été intégré à Prolog III.

En ce qui concerne les nombres flottants, leur représentation interne, et donc leur précision et leur étendue , sont déterminés par les particularités de votre machine. Leur syntaxe coïncide avec celle utilisée dans la plupart des langages qui connaissent ce type de données numériques. Elle est donnée en détail dans le chapitre consacré à la syntaxe

Si les nombres flottants sont fort utiles, voire indispensables, dans la résolution de nombreux problèmes, on peut aussi considérer qu'ils ont des effets nuisibles, notamment sur l'exactitude des calculs dans lesquels ils interviennent. De plus, le statut des nombres flottants parmi les autres objets qui constituent le domaine de Prolog III peut paraître au premier abord assez confus. Il y a donc lieu de donner tout de suite quelques explications, et pour commencer sur ce que les flottants ne sont pas :

• Les nombres flottants ne sont pas une notation alternative pour les nombres rationnels. Il ne faut pas croire que `75.5` est une autre manière d'écrire `755/10.` Même si dans tel ou tel cas particulier l'opération est exacte, la nature des calculs mis en œuvre pour convertir un nombre flottant en un rationnel ne permet pas d'assurer un résultat généralement satisfaisant. Sans compter que pratiquer systématiquement de telles conversions serait trahir l'esprit même des nombres flottants, en leur accordant une exactitude qu'ils n'ont jamais possédée, ni même revendiquée.

• Les nombres flottants ne constituent pas un “autre” domaine numérique, disjoint de celui des nombres rationnels. Autrement dit, il n'est pas insensé de se demander si la contrainte `75.5 = 755/10` est satisfaite ou non. Plus encore, non seulement une telle contrainte a un sens, mais il faut savoir que Prolog III s'efforce de la traiter de la manière la plus précise possible ; ainsi, deux nombres dont l'un au moins est flottant sont trouvés égaux si leur écart est inférieur à un certain seuil qui prend en compte les nombres en question, les caractéristiques du codage des nombres flottants sur votre système hôte ainsi qu'un savant calcul d'erreur ; en outre, la contrainte d'inégalité (≠) est traitée de manière tout-à-fait cohérente avec la précédente.

Voici en quelques points la bonne manière d'envisager les ensembles de nombres que traite Prolog III :

1. Le domaine numérique est constitué par l'ensemble des nombres réels au sens mathématique du terme, comprenant en particulier les nombres rationnels, les nombres algébriques et les nombres transcendants, même si certaines de ces familles de nombres restent en pratique inaccessibles.

2. Les nombres qui ont été exprimés sous la syntaxe des rationnels représentent pour l'utilisateur des données exactes. Les quatre opérations arithmétiques (`+, -, *, /`) sont elles aussi qualifiées d'exactes. *Les résultats d'opérations exactes sur des nombres exacts sont exacts*. Finalement, il existe une syntaxe, celle des nombres rationnels, pour exprimer exactement la valeur des nombres exacts.

3. Pour des raisons d'efficacité et d'ouverture vers d'autres domaines d'application, on introduit une syntaxe permettant d'exprimer des nombres inexacts : la syntaxe des nombres flottants. La porte à l'inexactitude ayant été ouverte, on ajoute alors un certain nombre d'opérations inexactes, comme la racine carrée ou les fonctions trigonométriques et logarithmiques, dont l'utilité n'est plus à prouver.

4. L'inexactitude est contagieuse : le résultat d'une opération inexacte, ou d'une opération exacte sur des nombres inexacts est inexact.

5. Etre inexact n'est pas une infamie : Prolog III s'efforce de travailler avec les nombres inexacts de la manière la plus précise possible, comme nous l'avons déjà dit par exemple au sujet de `x = y` et `x ≠ y` lorsque `x` ou `y` sont des nombres flottants.

Expressions numériques

Les expressions numériques sont des termes construits à partir de variables Prolog III, de constantes numériques et d'opérateurs numériques.

© PrologIA

Les constantes numériques

Il y a deux types de constantes numériques en Prolog III. Ce sont les nombres entiers positifs ou nuls et les nombres flottants positifs ou nuls.

Les nombres entiers positifs ou nuls.

En voici quelques exemples :

```
0
1
2
1789
81591528324789773434561126959611589427200000000[1]
```

{ On rappelle que ni les entiers négatifs, ni les nombres fractionnaires ne sont considérés comme des constantes. On notera que l'on peut les exprimer comme résultats d'opérations sur des entiers non négatifs.

Les nombres flottants positifs ou nuls.

Exemples :

```
12.345
0.5
1.2E12
0.888e87
```

La syntaxe exacte pour les flottants est donnée au chapitre réservé à la syntaxe

Les opérateurs numériques

il s'agit des opérateurs suivants :

+	:	plus unaire
-	:	moins unaire
+	:	addition
-	:	soustraction

1 Le lecteur attentif aura reconnu là le résultat de 40!

`/` : division
`*` : multiplication

{ Syntaxiquement, Prolog III, conservant en cela une certaine philosophie de la notation mathématique, autorise d'omettre le signe de la multiplication lorsque cela ne génère aucune ambiguïté. Par exemple, le système de contraintes `{X+2Y = 1,3Z-2T < 5}` est parfaitement correct. Par contre la contrainte `{(X+Y)(X-Y) # 0 }` est elle aussi correcte, mais signifie que l'arbre dont l'étiquette est la valeur `X+Y`, et dont l'unique fils est la valeur numérique `X-Y` est différent de l'arbre réduit à la feuille `0`, ce qui, d'ailleurs, est toujours vérifié.

Priorités des opérateurs numériques

Afin de permettre la suppression d'un certain nombre de parenthèses, on introduit un ordre de priorité sur les opérateurs numériques :

- Les opérateurs `*` et `/` sont prioritaires sur les opérateurs `+` et `-`

Tout d'abord on notera que les opérateurs plus et moins unaires ont le même ordre de priorité que les opérateurs plus et moins binaires. Ensuite, en cas d'égalité des ordres de priorités, l'opérateur de gauche est prioritaire par rapport à celui de droite.

Exemple :

`((x*y)/z)+(x*(y/z))` s'écrit plus simplement : `x*y/z+x*(y/z)`

Voici à présent un certain nombre d'expressions numériques correctes (pour la syntaxe exacte des expressions numériques on se reportera au chapitre Syntaxes) :

(1) `1993/1989`

(2) `+X`

(3) `(3/2)X`

(4) `2X+5Y`

(5) `3/2-X+7.02`

© PrologIA

(6) `X+2Y/Z`

et quelques unes incorrectes

(7) `2(X)`

(8) `X2`

(9) `3+-X`

(10) `3/2X`

(11) `X = 2Y+1`

Voici quelques remarques concernant ces exemples :

(2) est une expression numérique qui impose simplement à **x** de représenter une valeur numérique. On remarquera en passant que le terme **x** n'est pas une expression numérique, la variable **x** représentant un arbre quelconque.

(3) L'expression est parenthésée, et n'admet pas l'omission des parenthèses (voir l'exemple (10) et le chapitre sur la syntaxe des expressions numériques, particulièrement en ce qui concerne l'omission possible de l'opérateur de multiplication. On se rappelera à ce sujet que cet opérateur peut être omis uniquement dans les expressions de type *Constante*Variable* et *(Expression Numérique)*Variable* .

(6) est une expression numérique correcte, mais non-linéaire. Le traitement correspondant, et plus particulièrement celui des contraintes faisant intervenir de telles expressions est particulier et sera détaillé dans les chapitres qui suivent.

(7) est un terme correct, qui représente un arbre étiqueté par la valeur **2** et dont le fils unique est représenté par la variable **x**. Ce n'est donc en aucun cas une expression numérique.

(8) est un terme formé d'une unique variable d'identificateur **x2**, et pas un produit de la variable **x** par la valeur **2**.

(9) et (10) exigent un parenthésage (voir chapitre Syntaxes).

(11) n'est pas une expression numérique, même pas un terme Prolog III. C'est tout de même un objet syntaxique du langage, puisque c'est une contrainte numérique. On veillera cependant à ne pas confondre ces deux types d'objets.

Les relations

On trouve, pour la partie numérique de Prolog III deux types de relations : les relations unaires, qui serviront à préciser le caractère numérique de certains arbres, ce qui revient, dans les faits, à formuler des contraintes de type, et les relations binaires, utilisées pour construire des contraintes numériques proprement dites. Voici les différentes relations qui permettent de construire des contraintes numériques :

Relations unaires.

`X !numt` qui impose à la variable **`x`** de représenter un arbre étiqueté par un nombre réel ou rationnel. (Syntaxiquement, on remarquera qu'il n'est pas autorisé d'insérer, comme dans toutes les relations unaires de typage un espace entre le ! et numt)

`X !num` qui est équivalent à l'association des deux contraintes **`X !numt`** et **`X :: 0.`**

Relations binaires.

`=`	:	égalité
`#`	:	inégalité
`<`	:	inférieur strict
`>`	:	supérieur strict
`<=`	:	inférieur ou égal
`>=`	:	supérieur ou égal

© PrologIA

Contraintes numériques

Les contraintes numériques sont des objets syntaxiques qui expriment, dans le cas des relations binaires, la mise en relation de deux expressions numériques. L'utilisation de contraintes unaires induit, elle, un typage des variables considérées.

A propos de types, on notera la propriété suivante :

{ la seule présence d'une variable dans une expression numérique contraint celle-ci à représenter une valeur numérique.

Par exemple la contrainte `{X = Y+2Z}` est en fait codée sous la forme suivante : `{ Y !num, Z !num, X = Y+2Z}.` le terme figurant à gauche de l'égalité, `X,` n'étant pas une expression numérique, aucun typage particulier n'est effectué et, du point de vue du typage, la variable `X` représente un arbre quelconque, même si pour vérifier cette égalité, `X` est contraint à représenter un nombre.

Exemples de contraintes numériques

Voici quelques exemples de contraintes numériques correctes :

- `2X-3 = 0`
- `X+5 = 3*(2X-1)+2`
- `X # 1+Y`
- `X > Y`
- `1 <= 4X-1 <= 9`

Restrictions

La principale restriction portant sur les contraintes numériques réside dans le fait que, pour des raisons d'efficacité évidentes, seules les équations et inéquations linéaires sont prises en compte par les algorithmes de résolution de Prolog III. De ce fait, le traitement de toute contrainte contenant une multiplication de deux variables ou une division par une variable est retardé, jusqu'à ce que le nombre de variables connues soit suffisant pour rendre la contrainte linéaire. Dans la majorité des cas cette linéarité sera obtenue dès l'unification. Nous examinerons en détail le processus mis en place dans les autres cas dans un chapitre spécialement consacré à ce sujet.

Une autre restriction importante est que l'on ne peut, en Prolog III, imposer à un terme de représenter une valeur entière. Une façon de traiter les entiers consiste à définir des ensembles de contraintes portant sur des nombres rationnels et, en fin de traitement, d'effectuer une énumération entière des valeurs possibles de ces variables grâce à la règle prédéfinie **`enum`** (voir le chapitre "Règles prédéfinies et procédures externes")

Forme normale

Nous évoquons ici la forme normale utilisée pour le codage les contraintes numériques afin d'apporter certains éclaircissements concernant les expressions rencontrées au cours des sorties (lors de l'utilisation du prédicat évaluable **`list`** par exemple).

Les équations

Tout d'abord examinons la forme normale utilisée pour le codage des équations. Dans une égalité entre des expressions numériques, une variable est privilégiée (c'est une variable dont le domaine général des valeurs possibles englobe le domaine général des valeurs possibles des autres variables. On choisira, dans l'ordre, une variable de type arbre, numérique non contrainte, numérique contrainte à représenter une valeur positive ou nulle). Cette variable (dans le cas ou plusieurs variables sont candidates, le choix est arbitraire, ou plus précisément non maîtrisable par l'utilisateur) est alors exprimée en fonction des autres variables.

© PrologIA

Pour visualiser cette mise sous forme normale, interrogeons l'interpréteur :

```
> { 1X = (1/2)Y+2Z-4 };
  { Z = -(1/4)Y+(1/2)X+2 }

> { X = (1/2)Y+2Z-4 };
  { X = 2Z+(1/2)Y-4 }

> { 1X = (1/2)Y+2Z-4,Z >= 0 };
  { Y = 2X-4Z+8, Z >=0 }
```

Dans le premier exemple, les trois variables **X, Y** et **Z** sont des variables numériques non-contraintes.

Dans le second exemple, **X** est une variable de type arbre quelconque, les deux variables **Y** et **Z** sont des variables numériques non-contraintes.

Dans le troisième exemple, **Z** est une variable numérique contrainte à représenter une valeur positive ou nulle, les deux variables **X** et **Y** sont des variables numériques non-contraintes.

Les inéquations

Les inéquations rencontrées sont transformées de la manière suivante, en introduisant une variable supplémentaire α, appelée *variable d'écart :*

$X>Y$	est transformée en			$X-Y = \alpha,\ \alpha > 0$
$X \geq Y$	"	"	"	$X-Y = \alpha,\ \alpha >= 0$
$X<Y$		"	"	$Y-X = \alpha,\ \alpha > 0$
$X \leq Y$	"	"	"	$Y-X = \alpha,\ \alpha >= 0$

Cette transformation effectuée, la contrainte de type équation est mise sous forme normale, comme exposé plus haut.

Voici deux exemples de contraintes linéaires et la forme proposée en sortie par l'interpréteur :

```
> { X+3Y >= Z+2T };
  { T = -(1/2)Z+(3/2)Y+(1/2)X-(1/2)X', X' >= 0 }

> {0 <= (7/11)X+(4/6)Y};
  { Y = -(21/22)X+(3/2)X', X' >=0 }
```

Par souci de lisibilité, les variables d'écart créées par Prolog III ont été renommées.

3. Règles prédéfinies et procédures externes spécifiques

enum(N)

Cette règle prédéfinie permet d'énumérer toutes les valeurs entières que peut prendre le terme *N* dans le domaine de valeurs défini par l'ensemble de contraintes courant. Cette règle prédéfinie permet de réaliser des programmes dans lesquels interviennent des nombres entiers. Le principe de base, bien sûr, consiste à contraindre le terme auquel on s'intéresse de la manière la plus complète possible, ceci afin de minimiser le nombre de valeurs à énumérer.

{ Attention toutefois à l'ordre d'énumération des variables, des écarts de performances énormes pouvant en découler. En règle générale, on choisira d'énumérer d'abord les variables les plus contraintes. Si les contraintes sont identiques, on choisira d'abord celles dont les coefficients sont les plus élevés ou encore celles qui interviennent le plus souvent dans le système de contraintes.

On examinera comme exemple le cryptarithme SEND+MORE = MONEY, plus avant dans ce chapitre.

© PrologIA

max_value(R1,R2)
min_value(R1,R2)

Cette procédure externe à deux arguments calcule la valeur du maximum (respectivement du minimum) d'une expression numérique **R1**. Si on veut être plus précis, la valeur qui est calculée est, en fait, le plus petit des majorants (respectivement le plus grand des minorant), dans le cas de max_value (respectivement de min_value). Ceci implique que le maximum (respectivement le minimum) calculé n'est pas forcément atteint, ce calcul ne tenant pas compte des contraintes d'inégalités de type ≠, et des contraintes d'inégalités strictes (<, >).

particular_value(R1, R2) : Pose la contrainte **{R2 = R1'}**, où **R1'** est une valeur quelconque du terme **R1** qui satisfait l'ensemble de contraintes courant. Pour les remarques diverses concernant cette primitive, se reporter au chapitre règles prédéfinies et procédures externes).

mult(R1,R2,R3) pose la contrainte **{R3 = R1*R2}** avec les retardements nécessaires s'il y a lieu.

bound_mult(R1, R2, R3) pose la contrainte **{R3 = R1*R2}**, à la condition que la valeur de **R1** ou celle de **R2** soit connue, échoue dans le cas contraire.

Vérifications de types

integer(I) vérifie que le terme **I** représente une valeur entière connue.

num(R) vérifie que le terme **R** représente une valeur numérique connue.

rational(R) vérifie que le terme **R** représente une valeur numérique rationnelle connue.

real(R) vérifie que le terme **R** représente une valeur numérique codée en flottant connue.

is_num(T) vérifie que le terme **T** est astreint à représenter une valeur numérique, ou que la contrainte **{T !num}** appartient au système courant de contraintes.

Divers

trunc(R, N) pose la contrainte **{N = R'}**, où **R'** est la partie entière du nombre **R**. Echoue si **R** n'est pas une valeur numérique connue.

div(N1,N2,N3) pose la contrainte **{N3 = E}** où **E** est le résultat de la division entière de **N1** par **N2**. Seules les valeurs absolues de **N1** et **N2** sont prises en compte pour le calcul de **E**. Echoue si **N1** et **N2** ne sont pas tous les deux des valeurs entières connues.

mod(N1,N2,N3) pose la contrainte **{N3 = R}** où **R** est le reste de la division entière de **N1** par **N2**. Seules les valeurs absolues de **N1** et **N2** sont prises en compte pour le calcul de **R**. Echoue si **N1** et **N2** ne sont pas tous les deux des valeurs entières connues.

4. Retardement des contraintes non-linéaires

Nous examinerons dans le détail le processus de retardement des contraintes numériques non-linéaires dans un chapitre entièrement consacré aux retardements. Au niveau ou nous sommes, permettons nous tout de même une remarque. Prolog III autorise l'écriture de contraintes non-linéaires. Retenons dans un premier temps que Prolog III ajoute au système de contraintes numériques toute contrainte qui est linéaire au moment où on se propose de réaliser cet ajout. En d'autres termes, toute contrainte syntaxiquement non-linéaire est traitée de manière tout à fait naturelle si les valeurs de suffisamment de variables sont connues au moment de l'exécution.

Voici un exemple :

© PrologIA

```
> calculer(X,Y,Z)  ->
     {    Z = (3X-Y)/Z,   X+Y-2Z = 0    };
{}
> calculer(X,Y,2);
  { X = 2, Y = 2 }
> calculer(1,Y,Z);
  { Y = Y1, Z = (1/2)Y1+1/2,
    ((1/2)Y1+1/2)*((1/2)Y1+1/2) = -Y1+3,
    (1/2)Y1+1/2 # 0 }
```

Dans le premier cas, au moment de traiter l'ensemble de contraintes lié à la seule règle qu'il est possible d'appliquer pour répondre à la requête, celui-ci est linéaire. En effet, le simple fait de connaître la valeur de **Z** suffit à réaliser cette condition. On obtient donc une réponse qui fournit les uniques valeurs de **X** et **Y** qui satisfont ce système linéaire de deux équations à deux inconnues.

Dans le second cas, par contre, le nombre de valeurs connues ne suffit pas à rendre le système linéaire, et la réponse fournie par Prolog III donne certains renseignements sur les variables gelées lors du retardement de cette contrainte. Pour plus de précisions sur ce type de retardement se reporter au chapitre "Retardements"

5. Formats d'entrée-sortie

Les nombres utilisés en Prolog III se divisent, comme nous l''avons vu en deux classes : les rationnels en précision parfaite et les flottants. Pour améliorer le confort d'utilisation, Prolog III autorise un certain nombre de formats de sortie possibles. Ces formats sont positionnés grâce à la procédure externe **set_config(S,T).**

Le premier paramètre doit être une chaîne. Pour ce qui concerne les formats de sortie numérique **S** doit être égal à "**format_out_num**". Les valeurs possibles pour le second paramètre **T** sont alors :

- "**Fnormal**" : affiche les rationnels en précision infinie, et les flottants en notation scientifique
- "**Fexact**" : rationnels en précision infinie, les flottants sont convertis
- "**Fdecimal**" : affichage sous forme décimale
- "**Ffloat**" : affichage en notation scientifique
- "**Fint+fract**": affichage sous la forme partie entière + partie fractionnaire

On trouvera des exemples d'utilisation à la description de **set_config** au chapitre "Règles prédéfinies et prédicats évaluables"

6. Exemples de programmes

Calcul bancaire

Il s'agit de calculer la suite de versements successifs qu'il faut effectuer pour rembourser un capital emprunté à une banque. On suppose qu'entre deux versements s'écoule toujours le même temps et que pendant cette durée le taux d'intérêt réclamé par la banque est de 10%. L'ensemble des faits définis par le programme sera l'ensemble des arbres de la forme

versements_capital(*X*,*C*)

et où *X* représente la liste des versements nécessaires pour rembourser le capital *C* avec un intérêt de 10% entre deux versements. Le programme lui-même se résume à deux règles:

```
versements_capital(<>, 0) ->;
versements_capital(<V>.X, C) ->
   versements_capital(X, C+(10/100)C-V);
```

La première règle exprime qu'il n'est pas nécessaire de faire de versements pour rembourser un capital nul. La deuxième règle exprime que la suite des **n+1** versements pour rembourser un capital **c** consiste en un versement **v**

© PrologIA

et d'une suite **X** de **N** versements permettant de rembourser le capital **C** augmenté de 10% d'intérêts mais diminué du versement **V** effectué.

Ce programme peut-être utilisé de différentes façons. Une des plus spectaculaires est de demander pour quelle valeur de **V** la suite de versements **<V,2V,3V>** permet de rembourser 1000F. Il suffit de poser la requête

```
> versements_capital(<V,2V,3V>,1000);
  {V = 207 + 413/641}
```

Crypto-arihmétique

Le problème consiste à affecter aux 8 lettres *s,e,n,d,m,o,r,y* des chiffres tous différents de façon à ce que l'addition *send+more=money* tombe juste. La première façon de procéder consiste à profiter du fait que l'addition, la soustraction, la multiplication par une constante et les relations =,≠,<,≤,≥,> sont parfaitement connues au niveau des nombres rationnels.

On fait aussi intervenir le prédicat prédéfini **enum(X)** qui énumère tous les entiers **X** satisfaisant aux contraintes accumulées. Le programme est d'ailleurs purement déterministe jusqu'au moment où l'on appelle ce prédicat.

Voici le programme :

```
solution(I,J,K) ->
   chiffres_tous_differents(<S,E,N,D,M,O,R,Y>)
   tous_entiers(<M,S,O,E,N,R,D,Y>),
   { S # 0,M # 0,
     I =           1000 S + 100 E + 10 N + D,
     J =           1000 M + 100 O + 10 R + E,
     K = 10000 M + 1000 O + 100 N + 10 E + Y,
     I + J = K  };
```

```
tous_entiers(<>) ->;
tous_entiers(<X>.S) ->
   enum(X)
   tous_entiers(S);

chiffres_tous_differents(<>) -> ;
chiffres_tous_differents(<X>.S) -> ;
   hors_de(X,S)
   chiffres_tous_differents(S),
   {0 <= X <= 9};

hors_de(X,<>) ->;
hors_de(X,<Y>.S) -> hors_de(X,S),{X # Y};
```

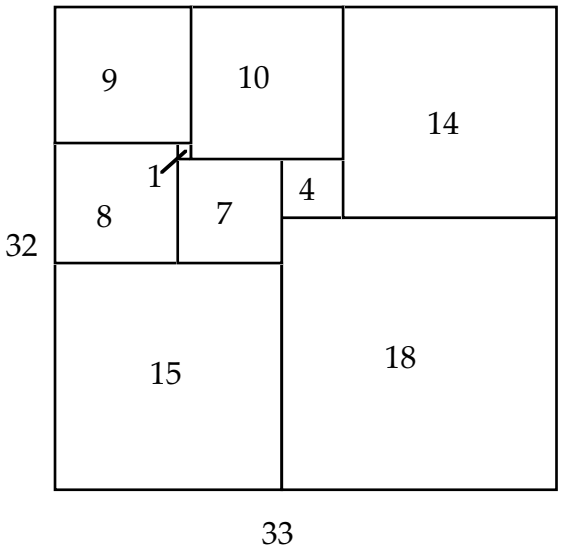
Si l'on pose la requête

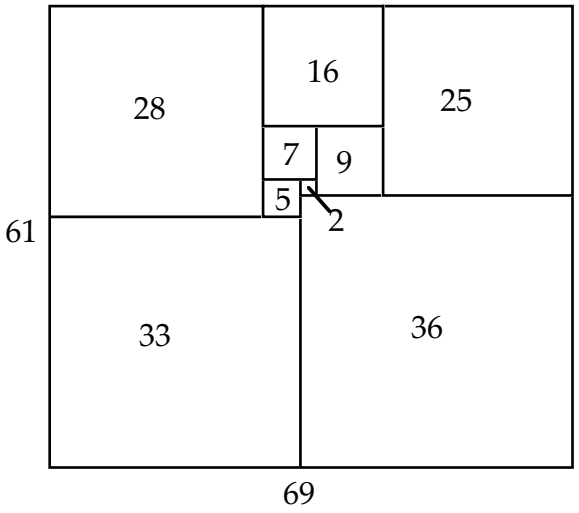
```
> solution(I,J,K);
  {I=9567, J=1085, K=10652}
```

Remplissage d'un rectangle par des carrés

Voici maintenant un problème, tiré de l'article d'Alain Colmerauer, "une introduction à Prolog III" qui met bien en valeur la partie numérique de Prolog III. Etant donné un entier **N** on s'intéresse à savoir s'il existe **N** carrés de dimensions distinctes qui peuvent être assemblés pour former un rectangle. Dans l'affirmative, on aimerait bien entendu connaître les dimensions de ces carrés et du rectangle formé. Voici par exemple deux solutions à ce problème pour **N=9**.

© PrologIA

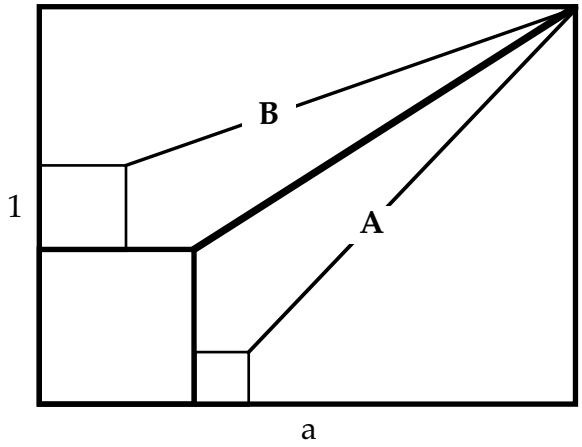




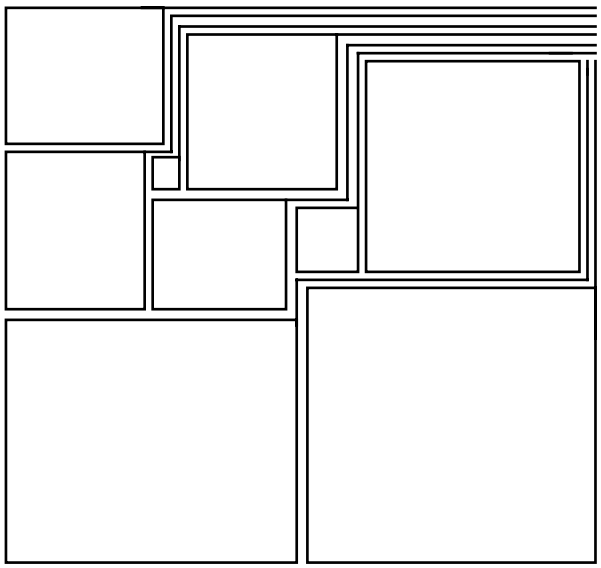
On désignera par **a** le rapport de la longueur du plus grand côté avec celle du plus petit côté du rectangle construit. On peut évidemment supposer que la longueur du plus petit côté est `1` et que la longueur du plus grand côté est **a**. Il faut donc remplir un rectangle de dimensions `1 x a` par `N` carrés tous distincts. Ce remplissage se fera en plaçant successivement chaque carré dans la position la plus basse possible et, à hauteur égale, dans la position la plus à gauche possible. En se référant au schéma qui suit, la base de l'algorithme de remplissage consistera alors

(1) à placer un carré dans le coin inférieur gauche du rectangle,
(2) à remplir de carrés la zone `A,` si elle n'est pas vide,
(3) à remplir de carrés la zone `B,` si elle n'est pas vide.

Le remplissage des zones `A` et `B` se fera récursivement de la même façon : placer un carré dans le coin inférieur gauche et remplir deux sous-zones.

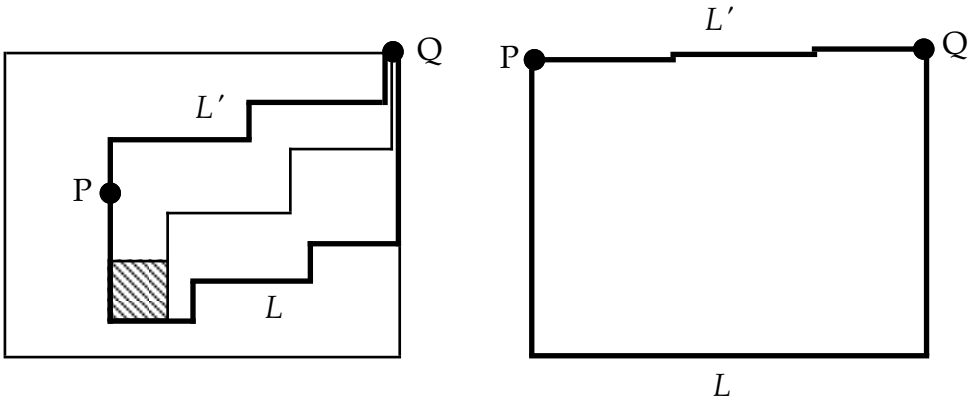


Les zones et les sous-zones sont séparées par des lignes brisées en forme d'escalier allant du coin supérieur droit des carrés au coin supérieur droit du rectangle. Ces lignes brisées ne descendent jamais et s'il est possible d'en tracer plusieurs pour aller d'un point à un autre on considère toujours la plus basse. Voici par exemple toutes les lignes de séparations correspondant à la première solution du problème lorsque `N = 9` :



© PrologIA

D'une façon plus précise une zone ou une sous-zone a la forme du schéma de gauche ci-dessous, la totalité du rectangle étant lui-même assimilé à la zone particulière dessinée à droite.



La zone est délimitée par une ligne brisée inférieure `L` allant d'un point `P` à un point `Q` et par une ligne brisée supérieur `L'` allant du même point `P` au même point `Q`. Le point `P` est placé n'importe où dans le rectangle à remplir alors que le point `Q` désigne le coin supérieur droit du rectangle. Ces lignes brisées sont représentées par des suite alternées de segments verticaux et horizontaux

$$v_0,\ h_1,\ v_1,\ \ldots\ ,\ h_n,\ v_n,$$

où `vi` désigne la longueur d'un segments vertical et h_i la longueur d'un segment horizontal. Les h_i sont toujours strictement positifs. Les v_i sont soit nuls, soit positifs pour désigner des segments qui montent, soit négatifs pour désigner des segments qui descendent. Les v_i des lignes supérieures ne sont jamais négatifs et si une zone n'est pas vide seul le premier segment vertical v_0 de sa ligne inférieure est négatif.

Si l'on applique ces conventions à la totalité du rectangle, (figure droite précédente) la ligne inférieure `L` peut être représentée par la suite `1,a,1` et la ligne supérieure `L'` par une suite de la forme $0,h_1,0,\ldots,h_n,0,$ avec $h_1+\ldots+h_n = a$, tous les h_i étant positifs.

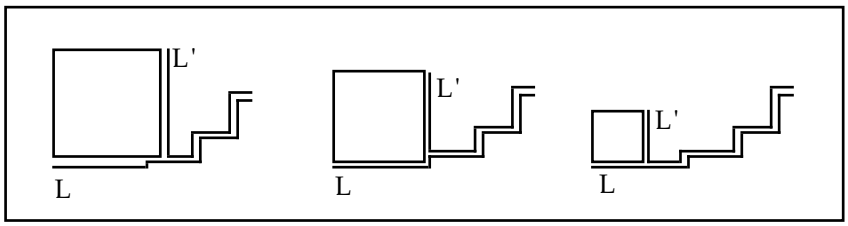
Le cœur du programme est la procédure

```
remplir_zone(L,L',C,C')
```

qui remplit de carrés une zone délimitée inférieurement par `L` et calcule sa délimitation supérieure `L'`. Les carrés sont puisés dans le début de la liste `C` et `C'` est la liste de carrés qui restent. Cette procédure fait appel à la procédure

```
placer_carre(b,L,L')
```

qui place un carré de dimension b x b dans le coin inférieur gauche de la zone à remplir. Ici `L` désigne la ligne inférieure de la zone, mais de laquelle on a enlevé le premier segment vertical, et `L'` désigne la ligne descendant du coin supérieur droit du carré pour rejoindre et se prolonger dans le reste de la ligne `L`. La figure ci-dessous montre les trois cas qui se présentent. Soit le carré déborde sur la première marche, qui en fait était une fausse marche de hauteur nulle, soit le carré est collé contre la première marche, soit le carré n'est pas assez grand pour toucher la première marche.



Le programme lui même est constitué des dix règles suivantes :

© PrologIA

```
remplir_rectangle(A, C) ->
    creer_carres(C)
    remplir_zone(<-1, A, 1>, L, C, <>) ,
    { A >= 1 } ;

creer_carres(<>) -> ;
creer_carres(<B>.C) ->
    creer_carres(C)
    rendre_distinct(B, C),
    { B > 0 } ;

rendre_distinct(B, <>) -> ;
rendre_distinct(B, <B'>.C) ->
    rendre_distinct(B, C) ,
    { B # B' } ;

remplir_zone(<V>.L, <V>.L, C, C) -> ,
    { V >= 0 } ;
remplir_zone(<V>.L, L''', <B>.C, C'') ->
    placer_carre(B, L, L')
    remplir_zone(L', L'', C, C')
    remplir_zone(<V+B, B>.L'', L''', C', C'') ,
    { V < 0 } ;

placer_carre(B, <H, 0, H'>.L, L') ->
    placer_carre(B, <H+H'>.L, L') ,
    { B > H } ;
placer_carre(B, <H,V>.L, <-B+V>.L) -> ,
    { B = H } ;
placer_carre(B, <H>.L, <-B, H-B>.L) -> ,
    { B < H } ;
```

L'appel général se fait par la requête

```
> remplir_rectangle(a, C), {C::n};
```

où **n** est le nombre de carrés de tailles distinctes qui doivent remplir un rectangle. Le programme calcule la dimension **1 x a** du rectangle (**a≥1**) et la liste **C** des dimensions des **n** carrés. Ce calcul débute par l'exécution de la première règle, qui à la fois contraint **a** à être plus grand ou égal à **1**, crée **n** carrés (de tailles inconnues) tous distincts et lance le remplissage de la zone constituée par la totalité du rectangle. La ligne **L** constituant la délimitation

supérieure de cette zone est inconnue au départ, mais, compte tenu que cette ligne doit joindre deux points qui se trouvent à la même hauteur et qu'elle ne peut descendre, ce sera forcément une ligne horizontale représentée par un escalier dont toutes les marches sont de hauteur nulles. Si on lance la requête :

```
> remplir_rectangle(a, C), {C::9};
```

on obtient 8 réponses. Les deux premières que voici

```
{a = 33/32,
 C = <15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32>}
{a = 69/61,
C =
<33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61>}
```

correspondent aux deux assemblages que nous avons dessinés. Les 6 autres réponses décrivent des assemblages symétriques de ceux-ci.

ststs

© PrologIA

Association Prolog
HERITAGE

Les contraintes booléennes

1. **Introduction**
2. **Quelques définitions et remarques**
   - **Expressions booléennes**
   - **Priorités des opérateurs booléens**
   - **Contraintes booléennes**
   - **Remarques importantes**
   - **Ensembles de contraintes booléennes**
   - **Exemples de contraintes booléennes**
   - **Forme normale conjonctive**
   - **Affectations booléennes**
   - **Solution d'un ensemble de contraintes booléennes**
   - **Simplification de systèmes de contraintes**
   - **Règles prédéfinies spécifiques**
3. **Premiers exemples de programmes**
   - **Un ou sur deux variables**
   - **Un ou sur une liste**
   - **Au plus un de vrai**
   - **K éléments vrais dans une liste de n booléens**
4. **D'autres exemples**
   - **Pannes dans un additionneur binaire**
   - **Un puzzle logique**

Que trouve-t-on dans ce chapitre ?

Ce chapitre est consacré aux contraintes booléennes et à leur utilisation. Tout en demeurant accessibles au lecteur peu familiarisé avec l'algèbre de Boole et le calcul propositionnel, nous y présentons les contraintes booléennes au travers de leur représentation syntaxique, mais nous abordons également des problèmes comme la représentation des sorties et la forme normale. Le chapitre se poursuit par l'examen de divers exemples, en suivant une progression régulière dans les niveaux de difficulté rencontrés, pour terminer par un exemple proche d'une petite application de diagnostic de pannes sur un circuit logique.

Décembre 1990

© PrologIA

1. Introduction

L'un des domaines sur lequel on peut faire porter des contraintes en Prolog III est celui des booléens. Les constantes, les opérations et les relations qui y sont définies permettent d'exprimer toutes les formules du calcul propositionnel, et autorisent par là même une puissance d'expression bien supérieure à celle des clauses de Horn ou des règles de production. On est ainsi en mesure, pour ce domaine, de traiter de manière rigoureuse la disjonction et la négation.

Pourtant, cette puissance se paye. En effet, les algorithmes de résolution de contraintes booléennes sont exponentiels. On pourra donc rencontrer certaines difficultés (temps d'exécutions trop longs, voire inacceptables) lorsque l'ensemble d'équations booléennes à traiter devient trop complexe. Pour tenter de remédier à cet état de fait, en dehors des efforts accomplis pour améliorer les algorithmes dans les versions futures, on pourra souvent faire chuter par une programmation adaptée, parfois de manière spectaculaire, les temps d'exécution. C'est pourquoi nous fournirons, dans ce chapitre, des conseils de programmation destinés à minimiser les temps de calcul.

On se propose ici, après quelques brefs rappels relatifs aux contraintes booléennes, de décrire les spécifications du langage pour ce qui concerne cette partie, avant de présenter un certain nombre d'exemples de programmes destinés à illustrer l'utilisation de ce type de contraintes.

2. Quelques définitions et remarques

Nous allons tout d'abord rappeler ce que sont, en Prolog III, les expressions booléennes, puis revenir sur les notions d'affectation, et de solution d'un ensemble de contraintes booléennes. Nous terminerons par la présentation des simplifications, et plus précisément de l'équivalence sur un sous-ensemble de variables, concept que nous rencontrerons fréquemment par la suite.

© PrologIA

Expressions booléennes

- Soit *V* le sous-ensemble des variables qui représentent des valeurs booléennes

- Soit *B = {0', 1'}* l'ensemble des valeurs que peuvent prendre les éléments de *V*. Ce sont les représentations en Prolog III des constantes *faux* et *vrai*

- Soient, enfin, les opérateurs suivants :

  ~ (non)
  & (et)
  | (ou)
  => (implique)
  <=> (équivalent)

Les expressions booléennes sont définies comme suit :

- *0'* et *1'* sont des expressions booléennes,
- les variables booléennes sont des expressions booléennes,
- si *f* et *g* sont des expressions booléennes, alors :

  ~(f) ,
  (f) | (g) ,
  (f) & (g),
  (f) => (g),
  (f) <=> (g), sont des expressions booléennes

Il est clair, d'après ce qui précède, que toute expression booléenne est un *terme* Prolog III.

Priorités des opérateurs booléens

Afin de permettre la suppression d'un certain nombre de parenthèses, on introduit un ordre de priorité sur les opérateurs booléens :

- L'opérateur ~ est prioritaire sur |, &, => et <=>
- Les opérateurs |, & sont prioritaires sur => et <=>

Exemples :

- `(~(a)|b)<=>(c&~(d))` s'écrit plus simplement : `~a|b <=> c&~d`
- `(a|(b|c))|(~(d)` s'écrit plus simplement : `a|b|c|~d`

Voici d'autres exemples d'expressions booléennes correctes :

- `1'`
- `b` si b est une variable typée booléenne
- `~~a|1'`
- `a&~b' <=> ~a'|b`
- `(a|b)&(a'<=>(b&(a'|0')))`

Contraintes booléennes

Pour représenter des contraintes booléennes, on se donne d'autre part les symboles relationnels suivants :

Relations binaires :

`=` *(égal)*
`#` *(différent)*
`=>` *(implique)*

{ On remarquera que *implique* est aussi en Prolog III une *relation*. En tant qu'opération cela permet souvent de simplifier l'écriture des expressions mais on prendra garde de bien la discerner de la *relation* implique.

On peut également imposer à un arbre `b` d'être étiqueté par une valeur booléenne en utilisant la contrainte unaire suivante :

```
b !boolt
```

On peut enfin imposer à un arbre `b` d'être réduit à une feuille et étiqueté par une valeur booléenne en utilisant la contrainte unaire suivante :

```
b !bool
```


© PrologIA

{ Une *contrainte booléenne* est, soit la relation unaire qui impose à une variable de représenter un arbre étiqueté par un booléen, soit une relation entre deux expressions booléennes.

Remarques importantes

Tout d'abord, il est à noter que par sa définition même, une contrainte est ou n'est pas vérifiée, et qu'en particulier celle-ci ne renvoie pas une valeur booléenne. On prendra donc garde de ne pas utiliser des expressions pour des contraintes, et inversement. Ainsi l'expression *a&b* n'est pas une contrainte, de même qu'il n'est pas correct d'écrire *(a>0)&(b>0)=1'*. C'est là une des restrictions de Prolog III que de ne pas permettre de mélanger les différentes algèbres. On se souviendra, de manière générale, qu'une contrainte booléenne n'est valide que si elle contient *exactement* un symbole relationnel.

Bien qu'il n'y ait pas, au sens strict, de notion de type dans Prolog III, on remarquera également que l'on peut contraindre une variable *b* à être booléenne (c'est à dire à représenter un arbre étiqueté par un booléen et réduit à une feuille), sans aucune autre restriction, de quatre façons :

- en posant la contrainte `b !bool`
- en posant les deux contraintes `b !boolt,` et `b :: 0` (cette dernière ajoute la contrainte que `b` est une feuille)
- en faisant apparaître cette variable dans une expression booléenne. Celle-ci pourra figurer dans une contrainte, comme dans `{b|~b=1'}`[1], ou dans un terme, par exemple booléen `(b|0')`[2].
- en faisant figurer cette variable comme expression dans une contrainte de type `=>.` Ceci impose en effet que les deux termes en relation soient des expressions booléennes. Une écriture possible est : `0'=>b`[3], où la seule contrainte imposée à la variable `b` est de représenter une valeur booléenne.

---

1 On notera au passage que l'expression *b | ~b* a toujours pour valeur *1'*.

2 L'expression *b | 0'*, tout comme *b&1'* a toujours la même valeur que *b*.

3 La contrainte *0' => b* est également toujours vérifiée, pour toute valeur de *b*.

Dans cet ordre d'idées, on notera que si `a` est une variable booléenne, la contrainte `a#b` (l'arbre représenté par la variable `a` est différent de celui représenté par la variable `b`) n'impose en aucun cas à `b` de représenter une valeur booléenne. En particulier, sa simple non-appartenance au domaine des booléens suffit à vérifier la contrainte.

Ensembles de contraintes booléennes

Comme nous l'avons déjà vu dans le cas général, les systèmes de contraintes booléennes sont des ensembles dont toutes les contraintes doivent être vérifiées simultanément. On remarquera, cependant, qu'il existe souvent plusieurs écritures possibles, le connecteur *et* implicite liant ces contraintes pouvant apparaître comme opérateur explicite dans une contrainte équivalente.

Par exemple, on pourra écrire indifféremment :

- `{a=1',b=1',c=1'}` ou `{a&b&c = 1'}`
- `{a => c, c # a|b}` ou `{(~a|c)&~(c<=>a|b) = 1'}`

Toutefois, pour des raisons d'efficacité, afin d'éviter un pré-traitement trop coûteux, on aura souvent intérêt à préférer de grands ensembles de contraintes simples à des systèmes constitués de peu de contraintes complexes.

Exemples de contraintes booléennes

Voici quelques exemples de contraintes booléennes correctes, ainsi que des ensembles de contraintes qui leur sont équivalents :

`a | ~a = 1'`

Cette contrainte est toujours vérifiée mais impose à *a* de représenter une valeur booléenne. Ensembles de contraintes équivalents :

`{0' => a},{a = (~x)[<>]},{a # ~a},...`

© PrologIA

```
(a<=>b)&(b<=>c)  =  1'
```

Celle-ci pose une équivalence entre trois variables booléennes. Ensembles de contraintes équivalents :

```
{a!boolt,a::0,a=b,a=c},
{a=>b,b=>c,c=>a}, ...
```

```
a&b&c  =>a'|b'
```

Cette contrainte, que l'on pourra aisément rapprocher d'une clause propositionnelle, voire d'une règle de production de système expert, présente la particularité de comporter un *ou* à droite de l'implication, ce qui est généralement interdit dans les démonstrateurs classiques. Ensembles de contraintes équivalents :

```
{~a|~b|~c|a'|b'=1'},
{~a'&~b'  =>  ~a|~b|~c},  ...
```

Voici un tableau récapitulatif des opérations et relations utilisables pour l'algèbre de Boole :

Opérations	Relations
Utilisées pour construire des *expressions booléennes*	Utilisées pour construire des *contraintes booléennes*
~ non	= égal
& et	# différent
| ou	=> implique
=> implique	
<=> équivalent	

Forme normale conjonctive

Au premier abord, la forme normale utilisée pour le codage des contraintes booléennes au sein de l'interpréteur semble ne concerner en rien l'utilisateur de Prolog III. Il nous paraît toutefois important de la mentionner pour deux raisons principales.

La première est liée au fait que l'efficacité de la résolution est d'autant plus grande que les transformations à effectuer pour le codage sont moins im-

portantes, ces transformations s'avérant fort coûteuses. La seconde concerne les sorties qui, pour les contraintes booléennes, sont toutes imprimées sous cette forme.

La *forme normale conjonctive,* ou *forme clausale,* d'une formule, est une formule équivalente écrite sous la forme de conjonctions de disjonctions de littéraux, c'est à dire de la façon suivante :

$$(l_1 \text{ ou } l_2 \text{ ou}\ldots l_n) \text{ et}\ldots (l'_1 \text{ ou } l'_2 \text{ ou}\ldots l'_p)$$

où les l_i et les $l'j$ sont des *littéraux,* c'est à dire des formules du type v ou *non(v),* avec v appartenant à l'ensemble des variables, V.

On aura tout intérêt à écrire, lorsque c'est possible, des contraintes proches de ce formalisme, afin d'éviter des transformations de coût exponentiel. L'exemple typique d'une contrainte dont la transformation est pénalisante est une représentation d'une formule en forme normale disjonctive. Un exemple en est :

```
{(a&b&c)|(a'&b'&c')|(a''&b'')}
```

dont la forme normale conjonctive est :

```
{ a|a'|a''  =  1',   a|a'|b''  =  1',
  a|b'|a''  =  1',   a|b'|b''  =  1',
  a|c'|a''  =  1',   a|c'|b''  =  1',
  b|a'|a''  =  1',   b|a'|b''  =  1',
  b|b'|a''  =  1',   b|b'|b''  =  1',
  b|c'|a''  =  1',   b|c'|b''  =  1',
  c|a'|a''  =  1',   c|a'|b''  =  1',
  c|b'|a''  =  1',   c|b'|b''  =  1',
  c|c'|a''  =  1',   c|c'|b''  =  1'   }
```

Les sorties, y compris la visualisation des contraintes associées aux règles codées par l'interpréteur, sont affichées sous forme de clauses, en respectant les conventions suivantes :

© PrologIA

- Si tous les littéraux sont positifs (de la forme l_i), on écrit :
$$l_1 \mid l_2 \mid \ldots \mid l_n = 1'$$
- Si tous les littéraux sont négatifs (de la forme $\sim l_i$), on écrit :
$$l_1 \,\&\, l_2 \,\&\ldots\, \&\, l_n = 0'$$
- Si certains littéraux sont négatifs (de la forme $\sim l_i$) et d'autres positifs (forme l'_j), on écrit :
$$l_1 \,\&\ldots\, \&\, l_n => l_1 \mid \ldots \mid l_p$$

Pour visualiser la mise sous forme clausale, on peut examiner la requête et la réponse suivantes :

```
> {a<=>b  =  1'};
  {a => b, b => a}
```

L'interpréteur analyse l'ensemble de contraintes initial, le code sous forme clausale, en vérifie la solubilité et l'imprime sous sa forme codée, après l'avoir simplifié. Nous étudierons dans les tout prochains sous-chapitres ces notions de solubilité et de simplification.

Voici d'autres exemples de mise sous forme normale conjonctive d'ensembles de contraintes, que vous pouvez également tester sur votre interpréteur :

- `{(a&~b)|(c&~d)=1'}` devient `{a|c=1',d=>a,b=>c,b&d=0'}`
- `{a|b#c,c  !boolt,c::0}` " `{a&c=0',b&c=0',a|b|c=1'}`

Affectations booléennes

Voici une définition de l'*affectation booléenne,* que l'on pourra rapprocher de la notion d'interprétation en logique propositionnelle. De manière globale, on pourra considérer qu'on affecte à toutes les variables de V une valeur dans $\{0', 1'\}$.

Une *affectation booléenne* est un ensemble $X= \{x_1 := a_1, x_2 := a_2, \ldots\}$ où à toute variable booléenne x_i est associée une valeur a_i prise dans l'ensemble $\{0', 1'\}$

A partir de ces affectations on peut définir la valeur d'une expression booléenne pour une affectation *X* de la manière suivante :

Soient *v* une variable de *V*, *f* et *g* deux expressions booléennes, alors :

- *0'/X = 0'*
- *1'/X = 1'*
- *v/X = 1'* si *v:=1'∈X*, *v/X = 0'* si *v:=0'∈X*
- *~f/X = 0'* si *f/X = 1'*, *~f/X = 1'* sinon
- *(f|g)/ X = 0'* si *f/X=0'* et *g/X = 0'*, *(f|g)/ X = 1'* sinon
- *(f&g)/X = 1'* si *f/X=1'* et *g/X = 1'*, *(f&g)/ X = 0'* sinon
- *(f=>g)/X =0'*si *f/X = 1 et g/X, = 1'*, *(f<=>g)/X =0'* sinon
- *(f<=>g)/X =1'*si *f/X = g/X*, *(f<=>g)/X =0'* sinon

Ces valeurs correspondent, bien sûr, à celles données par les tables de vérité habituelles pour les opérations concernées.

Solution d'un ensemble de contraintes booléennes

Une affectation booléenne est *solution* d'un ensemble de contraintes si et seulement si toutes les contraintes de cet ensemble sont vérifiées lorsque l'on a remplacé les expressions qui y figurent par leurs valeurs pour cette affectation.

Dans le même esprit, un système de contraintes booléennes est *soluble* s'il admet au moins une affectation booléenne qui en soit solution, *insoluble* sinon.

Exemples :

- Le système `{a=>b,b=>c,c=>a}` est soluble et admet les deux solutions suivantes : *X={a:=1', b:=1', c:=1'}*
  et *X'={a:=0', b:=0', c:=0'}*
- Le système `{~a|b=1',a|c=1',a#b,b#c}` n'admet, lui, aucune solution et est donc insoluble.

© PrologIA

Simplification de systèmes de contraintes

Lorsque l'on parle de simplification de contraintes, cela recouvre généralement deux aspects très différents. Le premier aspect de la simplification est de rendre le système considéré le plus lisible possible, grâce à une forme normale agréable, mais surtout à la suppression du plus grand nombre de redondances, principalement pour en diminuer la taille. Le second aspect concerne l'élimination des variables inutiles.

Pour ce qui est du premier aspect, Prolog III ne supprime pas toutes les redondances dans les contraintes booléennes, mais garantit cependant que chaque fois que le domaine des valeurs possibles pour une variable est réduit à un élément, l'équation correspondante figure dans le système de contraintes. Examinons, par exemple, la requête suivante et la réponse fournie par l'interpréteur :

```
> {a|b|c = 1',a => b,b => c};
  {c = 1', a => b}
```

La variable `c` est contrainte de représenter la valeur `1'` dans toutes les solutions du système initial. L'équation est donc produite, et l'ensemble de contraintes simplifié, en remplaçant `c` par `1'` dans chacune des contraintes de la requête.

Examinons à présent le mécanisme de suppression des variables inutiles. Voici, tout d'abord, la définition de l'équivalence sur un sous-ensemble de variables.

Deux systèmes de contraintes sont *équivalents sur un sous-ensemble de variables* V si et seulement si pour toute solution X de l'un, il existe une solution X' de l'autre qui coïncide avec X sur V et vice versa.

En particulier, pour les sorties, on cherchera à exhiber un système équivalent sur un sous-ensemble V, généralement l'ensemble des variables figurant dans la requête, qui ne contienne que des variables de V. On parlera parfois à ce propos de *projection sur un sous-vocabulaire.* Les contraintes présentes

dans ce système fourniront les liens qui unissent les variables considérées dans le système courant.

La nécessité d'une telle fonctionnalité est double. Tout d'abord, il arrive et nous en verrons des exemples, que l'on ne s'intéresse qu'aux liens qui unissent certaines des variables sur lesquelles on a fait porter des contraintes (par exemple, quels sont les liens entre les différentes pannes possibles, dans une configuration donnée, dans un système expert). Ensuite, de manière transparente pour l'utilisateur, l'interpréteur crée constamment de nouvelles variables, et il serait dommage que celles-ci, apparaissant dans des contraintes, ne viennent perturber la clarté (si l'on peut dire) des ensembles de contraintes résultats.

Voici, pour finir, quelques exemples de simplification de systèmes :

- La simplification du système `{a|x=1',a|~x=1'}` sur le sous-ensemble *{a}* donne le système `{a=1'}`

  Cet exemple est relativement trivial puisque le système initial est déjà équivalent au système `{a=1'}`

- La simplification de `{a|x=b|y,a&x#b&y}` sur *{a, b, c}* fournit le système `{a|b=1'}`

  On remarque sur cet exemple que l'une des deux variables `a` ou `b` doit être égale à `1'`. En fait, aucune solution du système initial ne comporte à la fois les affectations *a:=0'* et *b:=0'*.

- Enfin, le système `{a=>b|x,x=a&c,y|b#x&c}` simplifié sur *{a, b, c}* est égal à `{a=>c|b,a&b&c=0'}`

  Pas de commentaires particuliers, sinon qu'il est assez difficile de le vérifier à la main. Essayez donc…

Règles prédéfinies spécifiques

Les règles prédéfinies spécifiques aux booléens sont peu nombreuses. Les voici succinctement énoncées (On retrouvera ces règles dans le chapitre exclusivement consacré aux règles prédéfinies et prédicats évaluables de Prolog III)

© PrologIA

bool(B) : Vérifie que le terme B représente une valeur booléenne connue.

is_bool(B) : Vérifie que le terme B représente une valeur booléenne ou que la contrainte {B !bool} appartient au système courant de contraintes.

3. Premiers exemples de programmes

Nous allons à présent examiner quelques programmes Prolog III qui utilisent des contraintes booléennes.

Un *ou* sur deux variables

Un premier prédicat, tout à fait élémentaire, calcule le *ou* de deux variables. Le voici :

```
OuSimple(b1, b2, b1|b2) ->;
```

Quelques exécutions nous donnent une idée plus précise du traitement effectué, lorsqu'on pose ce type de contraintes.

```
> OuSimple(1',0',b);
  { b = 1'}
```

Le résultat de *ou* appliqué à *vrai* et *faux* est *vrai.*

```
>  OuSimple(b1,0',b);
  {  b => b1,
     b1 => b }
```

Le résultat de *ou* appliqué à *faux* et à une variable a la valeur de cette variable (l'égalité est représentée ici par une double implication)

```
> OuSimple(b1,b2,0');
  { b1 = 0',
    b2 = 0' }
```

Enfin, pour que le résultat d'un *ou* appliqué à deux variables soit *faux,* il faut que les deux variables représentent la valeur *faux.*

Un *ou* sur une liste

Le calcul d'un *ou* sur une liste de booléens est un tout petit peu plus compliqué. En fait, on veut exprimer les contraintes équivalentes à l'expression "il y a au moins un élément *vrai* dans la liste", le second argument du prédicat *Ou* ayant la valeur de cette expression.
Voici le programme :

```
Ou(<>, 0') ->;
Ou(<b1>.L, b1|b2) ->
   Ou(L, b2);
```

La règle d'arrêt exprime qu'il n'y a pas au moins un élément vrai dans la liste vide (il n'y en a bien sûr aucun). L'autre procède récursivement, en calculant le *ou* sur la tête de liste et sur le *ou* de la queue de liste.

Quelques exécutions pour y voir un peu plus clair :

```
> Ou(<1',b1,0'>,1');
  { b1 !bool }
```

La liste comportant d'ores et déjà un élément dont la valeur est *vrai,* aucune contrainte ne porte sur b1, si l'on excepte le fait que b1 doit représenter une valeur booléenne.

© PrologIA

```
> Ou(<b1,b1,b1>,1');
  {  b1 = 1'}
```

Ici, pour qu'il y ait au moins un élément *vrai* dans une liste composée de trois fois la même variable, il faut bien sûr que cette variable représente la valeur *vrai.*

```
> Ou(<b1,b2,b3>,b);
  {  b => b1|b2|b3,
     b1 => b,
     b2 => b,
     b3 => b }
```

En laissant variables toutes les valeurs, le système de contrainte équivalent exprime l'égalité entre la valeur de *b* et celle du *ou* des éléments de la liste.

Au plus un de vrai

On complique quelque peu le problème. On veut imposer cette fois-ci qu'il y ait au plus une variable de vraie dans une liste.

L'écriture de ce prédicat n'est pas vraiment triviale, au sens ou il n'y a pas de correspondance immédiate entre la valeur de *au plus un de vrai* sur une liste de tête *b1* et de queue *L* et sur celle de la liste *L.*

Si l'on observe le problème de manière un peu plus précise, on se retrouve devant l'alternative suivante :

Soit la tête de liste est *vrai*, et le *ou* de la queue de liste doit être faux, soit la valeur de la tête de liste est *faux*, et il faut qu'il y ait *au plus un élément de vrai* dans la queue de liste. On peut alors proposer une première solution de la forme suivante :

```
AuPlusUnVrai(<>,1')->;
AuPlusUnVrai(<b1>.L, b)->
   Ou(L,b2)
   AuPlusUnVrai(L,b3)
   { b = (b1&~b2) | (~b1&b3) };
```

Voici quelques exécutions :

```
> AuPlusUnVrai(<1',b2,b3>,1');
  { b2 = 0', b3 = 0'}

> AuPlusUnVrai(<b1,0',b2>,b);
  {  b&b1&b2 = 0',
     b|b1 = 1',
     b|b2 = 1'}
```

Pas de commentaires particuliers pour le premier exemple. Dans le second, le système de contraintes équivalent exprime les affirmations suivantes :

- **b1, b2** et **b** ne peuvent pas être vrais en même temps. (dans le cas contraire, **AuPlusUnVrai** renverrai la valeur **1'** avec deux éléments vrais dans la liste)
- **b1** ou **b** est vrai (dans le cas contraire, **AuPlusUnVrai** renverrai la valeur **0'**, alors que deux des éléments de la liste sur les trois sont faux)
- **b2** ou **b** est vrai (même raison que précédemment)

On peut également écrire ce programme sans renvoyer la valeur de l'expression, cela permettant de l'optimiser. On peut en effet considérer le problème de la manière suivante. Pour avoir au plus un élément vrai dans une liste de tête **b1** et de queue **L**, la première contrainte à vérifier est que le *et* de la tête et du *ou* de la liste soit faux. Il faut cependant que cette contrainte s'applique également récursivement sur chacune des sous listes auxquelles on a enlevé la tête de la précédente.

Par exemple, si on considère une liste de cinq éléments on aura :

```
{ b1  &  (b2|b3|b4|b5)  =  0'
  b2  &  (b3|b4|b5)  =  0'
```


© PrologIA

```
b3 & (b4|b5) = 0'
b4 & b5 = 0'  }
```

Voici donc une autre écriture possible :

```
AuPlusUnVrai(L) -> OuSurAuPlusUnVrai(L, b);

OuSurAuPlusUnVrai(<>, 0') ->;
OuSurAuPlusUnVrai(<b1>.L, b1|b2) ->
   OuSurAuPlusUnVrai(L, b2)
   {b1 & b2 = 0');
```

Avec les mêmes exemples d'exécution que pour l'exemple précédent :

```
> AuPlusUnVrai(<1',b2,b3>);
  { b2 = 0', b3 = 0'}

> AuPlusUnVrai(<b1,0',b2>);
  {  b1&b2 = 0'}
```

K éléments vrais dans une liste de n booléens

Voici, enfin, un dernier exemple de manipulation de contraintes sur des listes de booléens. Dans le programme suivant, le prédicat **`Vrais(k,L,b)`** a été construit de façon à ce que la valeur booléenne de **`b`** soit celle de l'expression :

"la liste de booléens L contient exactement k éléments vrais"

```
Vrais(k, <>, 0') ->
   {k ≠ 0};
Vrais(0, <>, 1') ->;

Vrais(0, <b1>.L, ~b1&b2) ->
   Vrais(0, L, b2);

Vrais(k, <b1>.L, b) ->
   Vrais(k-1, L, b2)
   Vrais(k, L, b3),
   {k ≠ 0, b = (b1 & b2) | (~b1& b3)};
```

L'idée générale de ce programme est de considérer le problème de la manière suivante :

Il y a exactement **k** éléments vrais dans une liste de booléens de tête b_1 et de queue **L**, si et seulement si l'une des deux propositions suivantes est vraie :

(i) b_1 est vrai et il y a exactement **k-1** éléments vrais dans la liste **L**
(ii) b_1 est faux et il y a exactement **k** éléments vrais dans la liste **L**

L'utilisation du paramètre **b** permet, en outre, de rendre le programme entièrement déterministe si **k** et la taille de la liste **L** sont connus.

Voici un exemple d'exécution de ce programme :

```
> vrais(2,<X,Y,Z,T>,1');
  {  X & Y & Z = 0',
     X & Y & T = 0',
     X & Z & T = 0',
     Y & Z & T = 0',
     X | Y | Z = 1',
     X | Y | T = 1',
     X | Z | T = 1',
     Y | Z | T = 1'     }
```


© PrologIA

On notera que la réponse fournie, après simplification et élimination des variables inutiles du système de contraintes final, induit une formalisation différente du problème posé, que l'on peut exprimer de la manière suivante :

k booléens sont vrais parmi n
si et seulement si
toute disjonction de (n-k+1) éléments est vraie,
et toute conjonction de (k+1) éléments est fausse.

On remarquera également que l'exécution d'une question dans laquelle `k` n'est pas connu donne un ensemble de réponses correspondant aux valeurs possibles de `k`.

Il existe également une manière de programmer cet exemple en utilisant un simple appel récursif, ce qui le rend nettement plus efficace. Nous laissons au lecteur le soin d'écrire ce programme à titre d'exercice.

4. D'autres exemples

Nous allons ici présenter quelques exemples, de tailles un peu plus volumineuses, qui permettent d'apprécier l'utilisation des contraintes booléennes pour résoudre un certain nombre de problèmes.

Pannes dans un additionneur binaire

Il s'agit de détecter le ou les composants défectueux dans un additionneur qui calcule la somme binaire de trois bits x_1, x_2, x_3, sous forme d'un nombre binaire de deux bits y_1, y_2. Comme on peut le voir dans la figure suivante, ce circuit est formé de 5 composants numérotés de 1 à 5 : deux portes *et* (marquées Et), une porte ou (marquée Ou) et deux portes *ou exclusif* (marquées OuX). Trois variables `u1,u2,u3` ont été introduites de façon à représenter les sorties des portes 1, 2 et 4.

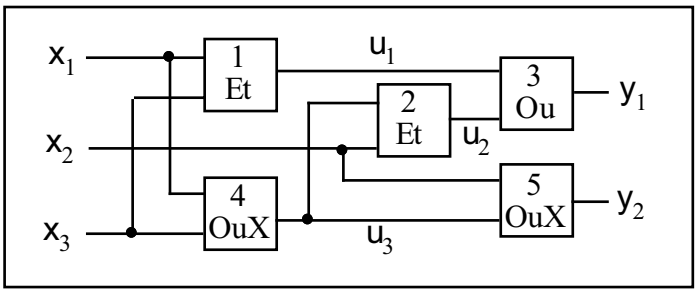


Fig. 4.1 *Le circuit*

On associe à chaque porte `i` une panne p_i et l'on se place dans une hypothèse de panne simple, c'est à dire que l'on admet que deux portes ne peuvent être défectueuses simultanément.

Voici les prédicats qui rendent compte de cette hypothèse. On reconnaîtra le prédicat **`au_plus_un_vrai`**, tel qu'il a été exposé précédemment.

```
au_plus_un_vrai(X) ->
   ou_au_plus_un_vrai(X, A);
ou_au_plus_un_vrai(<>, 0') ->;
ou_au_plus_un_vrai(<A>.X, A | B) ->
   ou_au_plus_un_vrai(X, B),
   { A & B = 0'  };
```

Le prédicat suivant décrit les liens qui unissent les pannes et les valeurs d'entrées-sorties du circuit.

```
circuit(<P1,P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>) ->
      au_plus_un_vrai(<P1,P2,P3,P4,P5>)
      { ~P1 => (u1 <=> X1 & X3),
            ~P2 => (u2 <=> X2 & u3),
            ~P3 => (Y1 <=> (u1 | u2)),
            ~P4 => (u3 <=> ~(X1 <=> X3)),
            ~P5 => (Y2 <=> ~(X2 <=> u3)) };
```


© PrologIA

Les contraintes utilisées décrivent le fait que si une porte n'est pas en panne, alors le résultat fourni par celle-ci est conforme aux prédictions que l'on est en droit de faire.

On peut se demander pourquoi on utilise ici une implication et pas une égalité (équivalence) entre les variables représentant les pannes pour une porte donnée et les expressions qui en expriment le bon fonctionnement. On opère ainsi pour traduire le fait qu'une porte peut être en panne et fournir, pour un jeu de données précis, une réponse correcte (l'exemple typique est celui d'une porte qui renvoie toujours la même valeur). Dans cette optique, le lancement du programme pour un jeu de données correct ne fournit pas comme résultat la valeur `0'` pour toutes les variables représentant les pannes, ces variables restant libres.

On peut utiliser ce programme de deux manières : soit en imposant des valeurs aux entrées et aux sorties du circuit, de façon à obtenir, le cas échéant, des informations sur les portes déficientes, soit en recherchant les jeux de tests qui permettent de caractériser une panne.

Voici tout d'abord deux requêtes qui recherchent une panne :

```
> circuit(<P1,P2,P3,P4,P5>,<1',1',0'>,<0',1'>);
  {P5=0',P4=1',P3=0',P2=0',P1=0'}
```

La porte 4 *(Oux)* est en panne.

```
> circuit(<P1,P2,P3,P4,P5>,<1',0',1'>,<0',0'>);
  {  P5=0',P4=0',P2=0',
     P3&P1=0',
     P3|P1=1'   }
```

Une seule des deux portes parmi *1* ou *3* est déficiente.

Intéressons nous à présent à rechercher les jeux de tests nécessaires à la caractérisation d'une panne donnée. La première idée est de lancer la requête en laissant inconnues les entrées et les sorties et en fixant une panne à vrai. Voici une telle requête, où l'on s'intéresse à la porte 1.

```
> circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>);

{  P2 = 0',   P3 = 0',
   P4 = 0',   P5 = 0',
   X3&X2 => Y1|X1,
   X2&X1 => Y1|X3,
   Y2&X3&X2 => X1,
   Y2 => X3|X2|X1,
   Y2&X2&X1 => X3,
   Y2&X3&X1 => X2,
   X2 => Y2|Y1,
   X2 => Y2|X3|X1,
   X3 => Y2|X2|X1,
   X3&X2&X1 => Y2,
   X1 => Y2|X3|X2     }
```

Si l'on excepte le fait que les autres pannes sont fausses, ce qui provient naturellement du prédicat **`AuPlusUnVrai`**, les renseignements fournis par le système simplifié sont plutôt illisibles. Une solution consiste à énumérer les solutions de ce système pour visualiser les différents jeux de données qui permettent de parvenir à ce résultat. Pour ce faire, on ajoute au programme les prédicats suivant, qui instancient les variables booléennes, de manière non déterministe.

```
booleans(<>)->;
booleans(<B>.L)->
   boolean(B)
   booleans(L);

boolean(0')->;
boolean(1')->;
```

On lance alors la requête en énumérant les solutions :

© PrologIA

```
> circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  booleans(<X1,X2,X3,Y1,Y2>);
  { X1 = 0',  X2 = 0',  X3 = 0',  Y1 = 0',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 0',  X3 = 0',  Y1 = 1',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 0',  X3 = 1',  Y1 = 0',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 0',  X3 = 1',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 1',  X3 = 0',  Y1 = 0',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 1',  X3 = 0',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 0',  X2 = 1',  X3 = 1',  Y1 = 1',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 0',  X3 = 0',  Y1 = 0',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 0',  X3 = 0',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 0',  X3 = 1',  Y1 = 0',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 0',  X3 = 1',  Y1 = 1',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 1',  X3 = 0',  Y1 = 1',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 1',  X3 = 1',  Y1 = 0',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
  { X1 = 1',  X2 = 1',  X3 = 1',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0'   }
```

On obtient ainsi toutes les solutions pour lesquelles il est possible que la porte 1 soit défectueuse. On peut cependant remarquer des solutions où le résultat est correct, par exemple la première, ainsi que d'autres pour lesquelles la défaillance de la porte 1 n'est pas la seule panne possible (on retrouve notamment à la dixième solution un des exemples précédents pour lequel l'une des portes 1 ou 3 était en panne).

On va donc tenter de raffiner encore le traitement pour n'obtenir plus que les jeux de tests nécessaires et suffisants pour isoler la panne. Il nous faut exhiber pour cela les solutions qui sont telles que seule la porte 1 est en panne.

Ce problème n'est pas spécialement trivial. Une idée pour le résoudre consiste à exhiber, non pas les jeux de tests qui nous intéressent, mais le complémentaire de ceux-ci parmi les solutions précédentes. Il suffit pour cela de ne retenir, parmi les solutions qui vérifient le système de contraintes pour

P1 = 1', celles qui vérifient également le système quand **P1 = 0'**. Voici la requête correspondante :

```
> circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  circuit(<0',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  booleans(<X1,X2,X3,Y1,Y2>);
```

Une autre solution envisageable consiste a se placer à un méta-niveau et à utiliser une négation par échec, puisque l'on veut connaître les données qui, d'une part vérifient le système de contraintes lorsque **P1** est vrai, et d'autre part qui ne vérifient pas celui-ci quand **P1** est faux.

Voici, tout d'abord le prédicat *non,* tout à fait classique en Prolog, qui s'efface si le prédicat **P** ne peut pas être effacé. :

```
non(P) -> P / fail;
non(P)->;
```

On peut alors poser notre requête de la manière suivante :

```
>  circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  booleans(<X1,X2,X3,Y1,Y2>)
  non(circuit(<0',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>));
```

Il faut noter ici que l'énumération doit se faire avant la négation, sous peine de rencontrer des problèmes, les variables n'étant pas encore instanciées (il est clair en effet dans ce cas que la négation sera systématiquement vérifiée, le système de contraintes avant instanciation des variables booléennes d'entrées-sorties étant soluble dans tous les cas). Or cette instanciation systématique «au plus haut» est fondamentalement en opposition avec la philosophie de Prolog III qui consiste à installer le plus de contraintes possibles avant toute énumération (voir le chapitre sur les contraintes numériques et le prédicat évaluable **enum**). Cette solution ne nous satisfait manifestement donc pas.

© PrologIA

On peut également, en restant à un méta-niveau, mais sans utiliser de négation par échec, utiliser le prédicat évaluable *known.* On rappelle à ce propos qu'une variable est *connue* si et seulement si elle représente un *arbre dont on connait l'étiquette initiale et dont on sait si le nombre de fils est nul ou non.*

Dans ce cas on pourra vérifier, pour chacune des solutions au système de contraintes pour lequel **`P1`** est *vrai,* que le système contraint cette dernière à représenter la valeur **`1'`** (c'est à dire qu'elle est connue) après un nouvel effacement du prédicat **`circuit`** où on reprend les entrées-sorties mais où la variable **`P1`** est cette fois-ci inconnue.

Voici cette dernière requête et le résultat fourni :

```
> circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  circuit(<P1,P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>)
  booleans(<X1,X2,X3,Y1,Y2>)
  known(P1);

  { X1 = 0',  X2 = 0',  X3 = 0',  Y1 = 1',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }

  { X1 = 0',  X2 = 0',  X3 = 1',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }

  { X1 = 0',  X2 = 1',  X3 = 0',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }

  { X1 = 1',  X2 = 0',  X3 = 0',  Y1 = 1',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }

  { X1 = 1',  X2 = 0',  X3 = 1',  Y1 = 0',  Y2 = 0',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }

  { X1 = 1',  X2 = 1',  X3 = 1',  Y1 = 0',  Y2 = 1',
    P2 = 0',  P3 = 0',  P4 = 0',  P5 = 0',  P1 = 1'    }
```

Cette solution n'est pas encore parfaitement satisfaisante. En effet, dans la mesure où les seules variables présentes dans le prédicat *Circuit* sont des variables booléennes, on peut imaginer de reporter directement la négation sur l'algèbre des booléens, et traiter par là même celle-ci de manière parfaitement rigoureuse.

Pour ce faire, il suffit d'ajouter un paramètre au prédicat **Circuit** et à **au_plus_un_vrai**, paramètre qui renvoie la valeur booléenne de l'expression qui représente le système de contraintes initial (on a déjà remarqué que l'on peut toujours transformer un système de contraintes booléennes en expression dont la valeur est *vrai).*

Voici le nouveau programme. On utilise la version de **au_plus_un_vrai** à trois arguments qui renvoie la valeur *b* de l'expression «Il existe au plus un élément vrai dans la liste **L**». Le résultat de **Circuit** est la conjonction de **b** et de la valeur de la nouvelle expression construite. Notons également au passage que les implications originales ont été transformées en disjonctions :

```
ou(<>, 0') ->;
ou(<B1>.L, B1|B2) ->
  ou(L, B2);

au_plus_un_vrai(<>,1')->;
au_plus_un_vrai(<B1>.L, B)->
  ou(L,B2)
  au_plus_un_vrai(L,B3)
  { B = (B1&~B2) | (~B1&B3) };

circuit(<P1,P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>,B1&B2) ->
  au_plus_un_vrai(<P1,P2,P3,P4,P5>, B1)
  { B2 = (P1|(U1 <=> X1 & X3)) &
    (P2|(U2 <=> X2 & U3)) &
    (P3|(Y1 <=> (U1 | U2))) &
    P4|(U3 <=> ~(X1 <=> X3))) &
    (P5 |(Y2 <=> ~(X2 <=> U3)))   };
```

On peut à présent faire porter la négation sur le dernier paramètre de **circuit** et la requête s'écrit de la manière suivante :

```
> circuit(<1',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>,  1')
  circuit(<0',P2,P3,P4,P5>,<X1,X2,X3>,<Y1,Y2>,  0')
  booleans(<X1,X2,X3,Y1,Y2>);
```

Le résultat est, bien sûr, identique à celui présenté précédemment.

© PrologIA

Un puzzle logique

Ce casse-tête logique, proposé par Lewis Caroll, a ceci d'intéressant qu'il est constitué d'une suite de phrases, que l'on peut aisément formaliser sous la forme de formules de logique propositionnelle, et qu'il ne comporte pas de questions. On peut donc s'amuser, à partir de ces phrases, à rechercher les liens qui existent entre certaines des propositions qui y figurent. C'est donc un exemple typique pour lequel la solution recherchée ne peut se présenter que sous la forme de la simplification sur un sous-ensemble de variables du système de contraintes initial. Voici les phrases proposées.

1. Tout individu apte à être député et qui ne passe pas son temps à faire des discours, est un bienfaiteur du peuple.
2. Les gens à l'esprit clair, et qui s'expriment bien, ont reçu une éducation convenable.
3. Une femme qui est digne d'éloges est une femme qui sait garder un secret.
4. Les gens qui rendent des services au peuple, mais n'emploient pas leur influence à des fins méritoires, ne sont pas aptes à être députés.
5. Les gens qui valent leur pesant d'or et qui sont dignes d'éloges, sont toujours sans prétention.
6. Les bienfaiteurs du peuple qui emploient leur influence à des fins méritoires sont dignes d'éloges.
7. Les gens qui sont impopulaires et qui ne valent pas leur pesant d'or, ne savent pas garder un secret.
8. Les gens qui savent parler pendant des heures et des heures et qui sont aptes à être députés, sont dignes d'éloges.
9. Tout individu qui sait garder un secret et qui est sans prétention, est un bienfaiteur du peuple dont le souvenir restera impérissable.
10. Une femme qui rend des services au peuple est toujours populaire.
11. Les gens qui valent leur pesant d'or, qui ne cessent de discourir, et dont le souvenir demeure impérissable, sont précisément les gens dont on voit la photographie dans toutes les vitrines.
12. Une femme qui n'a pas l'esprit clair et n'a pas reçu une bonne éducation, est inapte à devenir député.
13. Tout individu qui sait garder un secret et qui sait ne pas discourir sans cesse, peut être certain d'être impopulaire.
14. Un individu à l'esprit clair, qui a de l'influence et l'emploie à des fins méritoires, est un bienfaiteur du peuple.
15. Un bienfaiteur du peuple sans prétention n'est pas le genre de personnes dont la photographie est affichée dans toutes les vitrines.
16. Les gens qui savent garder un secret et qui emploient leur influence à des fins méritoires, valent leur pesant d'or.
17. Une personne qui ne sait pas s'exprimer, et qui est incapable d'en influencer d'autres, n'est sûrement pas une femme.

18. Les gens populaires et dignes d'éloges sont, soit des bienfaiteurs du peuple, soit des gens sans prétention.

Voici le programme. Un premier prédicat, **`Possibilité`**, lie les variables booléennes utilisées et les propositions présentes dans les phrases précédentes, et pose les contraintes portant sur ces variables.

```
Possibilite(<<a,"avoir l'esprit clair">,
  <b,"avoir reçu une bonne éducation">,
  <c,"discourir sans cesse">,
  <d,"employer son influence à des fins méritoires">,
  <e,"être affiché dans les vitrines">,
  <f,"être apte à être député">,
  <g,"être un bienfaiteur du peuple">,
  <h,"être digne d'éloges">,
  <i,"être populaire">,
  <j,"être sans prétention">,
  <k,"être une femme">,
  <l,"laisser un souvenir impérissable">,
  <m,"posséder une influence">,
  <n,"savoir garder un secret">,
  <o,"s'exprimer bien">,
  <p,"valoir son pesant d'or">>) ->
  { (f & ~c) => g,         (a & o) => b,
     (k & h) => n,         (g & ~d) => ~f,
     (p & h) => j,         (g & d) => h,
     (~i & ~p) => ~n,      (c & f) => h,
     (n & j) => (g & l),   (k & g) => i,
     (p & c & l) => e,     (k & ~a & ~ b) => ~f,
     (n & ~c) => ~i,       (a & m & d) => g,
     (g & j) => ~e,        (n & d) => p,
     (~o & ~m) => ~k,      (i & h) => (g | j)  };
```

Le reste du programme vérifie que la liste fournie en entrée est un sous-ensemble de la liste donnée dans **`Possibilite`**. Les contraintes sont alors mises en place, et l'algorithme de simplification sur un sous-ensemble de variables (voir les remarques concernant cette fonctionnalité) nous permet de visualiser les liens entre les variables du sous-ensemble considéré dans la question.

© PrologIA

```
SousPossibilite(x) ->
  Possibilite(y)
  SousEnsemble(x,y);

SousEnsemble(<>,y) ->;
SousEnsemble(<e>.x, y) ->
  ElementDe(e,y)
  SousEnsemble(x,y);

ElementDe(e, <e>.y) ->;
ElementDe(e, <e'>.y) ->
  ElementDe(e,y), {e#e'};
```

Intéressons nous dans un premier temps à établir les rapports éventuels qui existent entre "avoir l'esprit clair", "être populaire"et "savoir garder un secret". Il suffit pour cela de poser la requête suivante :

```
> SousPossibilite(<<p,"avoir  l'esprit  clair">,
      <q,"être  populaire">,
      <r,"savoir  garder  un  secret">>);
  {}
```

La réponse est l'ensemble vide de contraintes et nous signifie que suivant Lewis Caroll il n'y a aucun rapport entre "avoir l'esprit clair", "être populaire"et "savoir garder un secret".

Voyons à présent quels sont les liens qui unissent les propositions "savoir garder un secret","être apte à être député" et "valoir son pesant d'or". On pose la requête ci-dessous :

```
> SousPossibilite(<
  <p,"savoir  garder  un  secret">,
  <q,"être  apte  à  être  député">,
  <r,"valoir  son  pesant  d'or">>);
  {  p & q => r  }
```

Cette fois ci, la contrainte fournie en réponse exprime le fait que si l'on sait garder un secret et que l'on est apte à être député alors on vaut son pesant d'or.

ststs

© PrologIA

Association Prolog
HERITAGE

Retardements

1. **Introduction**
2. **Termes connus**
   - **Les prédicats évaluables known, bound et free**
3. **Retardement de l'exécution d'un but**
4. **Contraintes retardées**
   - **Tailles retardées**
   - **Concaténation retardée**
   - **Contraintes numériques retardées**

Que trouve-t-on dans ce chapitre ?

On retrouve en Prolog III le concept de retardement de l'exécution d'un but. La compréhension de ce processus demande que l'on s'attarde un peu sur la signification précise de ce qu'est une variable connue. De plus, un processus complexe de retardement de contraintes s'applique lorsque les restrictions de base sur la taille des arbres, la concaténation des tuples ou la linéarité des expressions numériques ne sont pas explicitement respectées. L'utilisation de ce processus de retardement de contraintes est un exercice qui peut s'avérer tout à fait périlleux, et nous essayerons dans ce chapitre d'en donner une explication détaillée.

1. Introduction

Comme Prolog II et Prolog II+, Prolog III permet de retarder l'exécution d'un but en attendant qu'un terme soit connu. La compréhension de ce

© PrologIA

concept demande que l'on s'attarde sur la signification du terme connu. De plus, certaines contraintes qui ne vérifient pas les restrictions imposées par Prolog III sont automatiquement retardées. Ce chapitre propose un tour d'horizon de ces différents retardements, ainsi qu'un certain nombre d'exemples. On conservera à l'esprit que la notion de contrainte retardée sort du cadre formel de Prolog III en fournissant un outil qui, s'il s'avère agréable à l'utilisation ouvre la voie à une programmation semée d'embûches.

2. Termes connus

La définition d'un terme connu est très précise en Prolog III :

{ Un terme est connu lorsque l'on connait l'étiquette de l'arbre qu'il représente et que l'on sait si son nombre de fils est nul ou non.

Souvent ces termes sont réduits à une variable et on parle alors de variable connue.

Les prédicats évaluables known, bound et free

Les trois prédicats évaluables suivants ont un rapport direct avec ce que l'on sait, au cours de l'exécution d'un programme, à propos d'un terme donné :

`known(T)` s'exécute avec succès si *T* est un terme connu au sens de la définition précédente

© PrologIA

bound(T) s'exécute avec succès si *T* est un terme représentant un arbre dont :

- L'étiquette initiale est connue
- Le nombre des fils est connu

free(T) s'exécute avec succès si l'on ne connait pas l'étiquette de l'arbre représenté par le terme *T*, et que l'on ne sait pas si le nombre de ses fils est nul ou non.

On remarquera au passage que **free(T)** n'est pas la négation de **known(T)**. Par exemple, **known(+x)** échoue car **+x** n'est pas "assez connu" (on sait que l'étiquette est un nombre, mais on ne connaît pas sa valeur), et **free(+x)** échoue aussi, car **+x** est "trop connu" (on sait que c'est une feuille).

3. Retardement de l'exécution d'un but

Prolog III permet de retarder l'exécution d'un but tant qu'un terme n'est pas connu (toujours au sens de la définition précédente). Aussitôt que celui-ci est connu, on tente d'exécuter le but retardé. Cette opération de retardement se réalise grâce au prédicat évaluable *freeze(X, P).*

Nous pouvons examiner ce traitement sur un exemple. Admettons que l'on désire construire le prédicat *creer(T1, T2, U)* qui retarde la création du tuple *U* formé des deux éléments *T1* et *T2* tant que l'on ne connait pas le premier de ces deux éléments. Voici les règles correspondantes :

```
creer(T1, T2, U) ->
  freeze(T1, creer'(T1,T2,U));

creer'(T1, T2, <T1, T2>)->;
```

On peut examiner une exécution de créer, qui mettra en évidence les sorties concernant les variables gelées.

```
> creer(illusion(A),T,U);
  { U = <illusion(A),T> }
```

Le premier paramètre étant connu (l'étiquette est connue et on sait si le nombre de fils, égal à un, est nul ou non), le prédicat *creer'* est immédiatement exécuté à l'appel.

```
> creer(T1,T2,U);
  { T1 = E[X],
    E[X] !freeze(creer'(E[X],T2,U)) }
```

Les variables intermédiaires créées par Prolog III ont été renommées par souci de lisibilité. Ici, le premier paramètre n'est pas connu à l'appel, et ne sera jamais connu jusqu'à la fin de l'exécution de la requête. En sortie, Prolog III rappelle que la variable *T1,* mise sous forme normale à l'aide du constructeur général d'arbres sous la forme *E[X]* demeure dans un état gelé, et rappelle quel est le prédicat retardé sur cette variable.

On peut compliquer un peu cet exemple et s'intéresser aux deux prédicats suivants :

- *creer_et(T1, T2, U)* qui retarde la création du tuple *<T1, T2>* jusqu'à ce que les deux termes *T1* et *T2* soient connus.
- *creer_ou(T1, T2, U)* qui retarde la création du tuple *<T1, T2>* jusqu'à ce que l'un des deux termes *T1* et *T2* soit connu.

Voici ces deux prédicats :

```
creer_et(T1, T2, U) ->
  freeze(T1, freeze(T2, creer'(T1,T2,U)));

creer_ou(T1, T2, U) ->
  freeze(T1, creer'(T1, T2, U))
  freeze(T2, creer'(T1, T2, U));

creer'(T1, T2, <T1, T2>)->;
```


© PrologIA

On remarquera que le prédicat *creer_ou* peut s'écrire d'une manière plus subtile afin d'éviter que *creer'* ne soit effacé deux fois. Une façon de résoudre ce problème est donnée dans le paragraphe consacré aux contraintes numériques retardées.

4. Contraintes retardées

Maintenant que le principe du retardement de l'exécution d'un but est défini, examinons comment Prolog III retarde certaines contraintes qui ne répondent pas aux restrictions imposées par le langage. C'est le cas :

- des contraintes de taille pour lesquelles cette taille n'est pas explicitement connue
- des contraintes faisant intervenir une concaténation pour laquelle la taille de l'opérande de gauche n'est pas connue
- des contraintes numériques non linéaires, c'est à dire faisant intervenir des produits de variables ou des divisions par une variable.

{ On essayera de ne pas perdre de vue que les contraintes qui ne respectent pas les restrictions de base de Prolog III sortent du cadre formel et ne devraient, en toute rigueur, figurer dans un programme. Le souci de faciliter la tâche du programmeur a conduit à admettre l'écriture de telles contraintes et à en définir un traitement approprié.

Dans ces trois cas, Prolog III automatise un certain type de retardement que nous allons exposer maintenant.

Tailles retardées

On sait (voir chapitre "Arbres, tuples, listes et chaînes") que dans une contrainte de taille, celle-ci doit être explicitement connue, c'est à dire figurer sous la forme d'une constante entière positive. On sait également que la règle prédéfinie *bound_size(U, N)* permet de poser une telle contrainte au cours de l'exécution d'un programme si la valeur de *N* est connue au moment où l'on tente d'exécuter le prédicat *bound_size.* Enfin la syntaxe de Prolog III permet d'écrire directement des contraintes du type *{U :: N}* ce qui peut sembler en totale contradiction avec les affirmations précédentes.

En fait, ce dernier type de contraintes est une facilité de syntaxe. Lorsque Prolog III doit traiter une contrainte de ce type, il tente d'exécuter, *avant* les autres buts de la règle où apparait cette contrainte, le prédicat *size(U, N)* qui est un prédicat prédéfini et qui utilise notamment *freeze, bound_size* et *known_part* (voir ces prédicats dans le chapitre règles prédéfinies et procédures externes).

Voici un premier prédicat, très proche du prédicat *size* :

```
delayed_size(T, N) ->
   freeze(T, delayed_size'(T, N))
   freeze(N, bound_size(T, N))

delayed_size'(E[<>], 0) ->;
delayed_size'(E[U], N) ->
   known_part(U, U', U'')
   bound_size(U', N')
   delayed_size(U'', N-N'),
   { U # <>, N # 0 };
```

On fera deux remarques à propos de ce prédicat :

- On profite du fait que la taille d'un terme est égale à celle du tuple formé de ses fils.
- On profite de tout renseignement sur le début de ce tuple pour ne retarder que les contraintes portant sur la partie de taille inconnue.

Examinons quelques exemples triviaux (on a remplacé les variables intermédiaires de Prolog III par des noms plus lisibles) :

```
> delayed_size(T,N);
  {  T = E1[U1], N = E2[U2],
     E1[U1] !freeze(delayed_size'(E1[U1],E2[U2])),
     E2[U2] !freeze(bound_size(E1[U1],E2[U2])) }
> delayed_size(Arbre[<1,2>.U],N);
  {  U !freeze(delayed_size'(U,N-2)),
     N !freeze(bound_size(Arbre[<1,2>.U],N)),
     N !freeze(bound_size(U,N-2))  }
>
```


© PrologIA

Nous pouvons dés à présent entrer un peu plus dans le détail. En fait Prolog III effectue un traitement plus élaboré que ce simple retardement. Voyons sur un exemple :

```
> {U :: N, U :: M, N # M};
>
```

Si tout se passait réellement comme nous l'avons décrit précédemment, les deux contraintes de taille se contenteraient de rester gelées, et l'incohérence ne pourrait en aucun cas être détectée (rappelons encore une fois qu'il ne s'agit pas là de *résolution,* mais de retardements). Prolog III permet toutefois de communiquer un certain nombre d'informations des arbres vers les variables numériques qui représentent leurs tailles. Dans le sens inverse (des tailles vers les arbres), ceci est généralement beaucoup plus difficile et n'est pas effectué dans la majorité des cas.

Examinons pour finir un exemple ou l'incohérence n'est pas détectée :

```
> { U :: N, 0 < N < 1 };
  { U = E[X],
    X :: N,
    -N+1 > 0,
    N > 0 }
>
```

De manière triviale, le système de contraintes donné en entrée n'admet aucune solution, puisque aucune valeur de N dans l'intervalle]0,1[n'est entière. Le processus de retardement ne permet pourtant pas de réaliser l'échec auquel on aurait pu s'attendre.

Concaténation retardée

La seconde restriction concerne les concaténations de tuples. Le retardement de ces contraintes intervient lorsque l'on a écrit une concaténation dont l'opérande de gauche est de taille inconnue

De la même manière que pour les contraintes de taille retardées, on peut donner une première explication satisfaisante du mécanisme mis en place dans ce cas en écrivant les règles Prolog III correspondantes. On rappelle que la règle prédéfinie qui met ce retardement en place est *conc3(U,V,W).* Voici un prédicat qui constituera une première approche de la concaténation retardée :

```
delayed_conc(U, V, W) ->
   freeze(U, delayed_conc'(U,V,W));

delayed_conc'(<>, V, V) ->;
delayed_conc'(U, V, W) ->
   known_part(U, U', U'')
   bound_conc(U', W', W)
   delayed_conc(U'', V, W'),
   { U # <> };
```

On profite, comme pour les contraintes de taille, de toute information partielle sur le début du tuple *U* pour dégeler ce qui peut l'être.

De la même manière que précédemment, on peut aller un peu plus loin, en tenant compte du fait que la taille du tuple résultat est égale à la somme des tailles des tuples concaténés. Voici un second prédicat, écrit dans cette idée, qui reproduit exactement le prédicat *conc3 :*

```
delayed_conc2(U, V, W) ->
   freeze(U, delayed_conc'(U,V,W))
   size(U, N)
   size(V, M)
   size(W, N+M);
```

Le prédicat *delayed_conc'* est le même que celui présenté plus haut.

Ainsi toute contrainte du type *W = U.V,* où la taille de *U* n'est pas connue, est remplacée, avant le traitement du premier littéral de la règle contenant cette contrainte, par l'appel d'un prédicat équivalent à *delayed_conc2(U, V, W).*

© PrologIA

On peut, bien sûr, s'apercevoir que cette manière de faire ne garantit nullement la complétude du traitement dans le cas où les restrictions ne sont pas respectées mais constitue bien un palliatif propre à éviter au programmeur la gestion de ce type de retardement.

Voici, à présent un certain nombre d'exemples de contraintes de concaténation retardée. Examinons, dans un premier temps, des contraintes très simples, et les réponses fournies par l'interpréteur :

```
> { U.V = W };
  { W = X,
    U :: N1,
    U.V = X,
    V :: N2,
    X :: N2 +N1,
    N1 >= 0 ,
    N2 >= 0 }
```

Nous avons ici le cas le plus général de contrainte retardée faisant intervenir des concaténations. La concaténation est retardée, et un certain nombre de contraintes de taille, ainsi que des contraintes numériques liant ces tailles sont ajoutées au système courant. Il est bon de se souvenir que ces contraintes de taille sont également retardées.

```
> { U :: N,
    V :: M,
    W :: P
    W = U.V,
    P # M+N };
>
```

```
> {U.X = V.X, U#V};
  { U :: N1,
    U.X = X4,
    U # V,
    X :: N2,
    V :: N1,
    V.X = X4,
    X4 :: N2+N1,
    N1 >= 0,
    N2 >= 0}
>
```

Cet exemple montre les limites des contraintes retardées faisant intervenir des opérations de concaténation. Dans ce cas de figure, les valeurs représentées par U et V ne sont pas suffisamment connues pour permettre de détecter l'incohérence de cet ensemble de contraintes.

Contraintes numériques retardées

Bien que seules les contraintes linéaires soient prises en compte au niveau des algorithmes fondamentaux de Prolog III, le langage autorise le programmeur à exprimer des contraintes non linéaires. Dans la plupart des cas il s'agira d'une facilité d'écriture pour éviter l'emploi du prédicat évaluable *bound_mult,* les expressions étant linéaires, de fait, au moment de l'unification, un nombre suffisant de variables présentes dans ces expressions étant alors connu. Dans le cas contraire, l'ajout de la contrainte est retardé, jusqu'à ce que celle-ci devienne linéaire.

Le processus, automatique, de retardement peut se programmer grâce à l'utilisation de *freeze,* et nous en expliquerons comme précédemment le fonctionnement en décrivant les règles Prolog III qui effectuent ce retardement des contraintes.

Voici ces règles, qui reproduisent le traitement effectué par la règle prédéfinie *mult.* La règle pour la division retardée est donnée pour mémoire :

```
delayed_mult(X, Y, Z)->
   freeze(X, delayed_mult'(X, Y, Z, A))
```


© PrologIA

```
      freeze(Y, delayed_mult'(Y, X, Z, A));

   delayed_mult'(X, Y, Z, A)-> known(A) /;
   delayed_mult'(X, Y, Z, 1')->
      bound_mult(X, Y, Z);
```

```
   delayed_divide(X, Y, Z)->
      delayed_mult(Z, Y, X)
      { Y#0 };
```

Dans le cas de la multiplication, il faut retarder le traitement de la contrainte jusqu'à ce que l'une *ou* l'autre des variables *X* et *Y* soit connue. On installe donc un double *freeze* sur les deux variables *X* et *Y*.

Le prédicat *delayed_mult'* permet, grâce à l'utilisation d'une variable de communication *A,* de ne pas exécuter *delayed_mult'* deux fois dans le cas où il existe une étape où les deux variables sont connues. En effet, la première exécution de *delayed_mult'* échouera sur la première règle, *A* n'étant pas connu, et après mise en place de la contrainte, affectera *A* à *1',* forçant par là même les appels suivants à exécuter la première règle, qui n'installe pour sa part aucune contrainte.

La division est construite naturellement en utilisant le prédicat *delayed_mult*.

Concernant le traitement retardé des contraintes numériques non linéaires, on retiendra, pour finir, la définition informelle suivante :

Dans le traitement des expressions numériques non-linéaires, tout se passe comme si toute contrainte de la forme *Z = E*E', où E et E'* sont des expressions contenant des variables, était transformée en un appel du prédicat *delayed_mult(E, E', Z).* De manière similaire, toute contrainte de la forme *Z = E/E',* où *E'* est une expression contenant des variables, est remplacée par un appel de *delayed_divide(E, E', Z).*

{

Cette façon de faire garantit notamment le respect d'un parenthésage éventuel. Par exemple les systèmes de contraintes *{X=A*(B+C)}* et *{X=A*B+A*C}* ne sont pas traités exactement de la même façon. Dans le second cas, si la valeur de *A* est inconnue, le fait de connaître *B* ou *C* dégèle une partie de la contrainte, alors que dans le premier cas on ne fait rien tant que leur somme n'est pas connue.

Examinons ce processus de retardement sur un exemple :

```
non_lineaire(X,Y,Z,T,V) ->
   {Z = X*Y+2T/(X+(3/2)V)};
```

Cette règle est considérée par le système de la manière suivante :

```
non_lineaire(X,Y,Z,T,V) ->
   delayed_mult(X,Y,X')
   delayed_divide(2T,X+(3/2)V,Y')
   {Z = X'+Y'};
```

En fait, la règle prédéfinie est *mult* que l'on a renommé pour que vous puissiez tester ces exemples sur votre interprète.

Voici un exemple d'exécution où l'on connait un nombre suffisant de valeurs pour rendre le système linéaire :

```
> non_lineaire(2,Y,Z,T,3);
  { Y = Y1, T = T1, Z = (4/13)T1+2Y1 }
>
```

Et un autre où des contraintes restent gelées :

© PrologIA

```
> non_lineaire(X,3,Z,T,2);
  { X = X', Z = U'+3X', T = T',
    X'+3 # 0,
    U'*(X'+3) = 2T' }
>
```

Comme on pouvait s'y attendre, il demeure un *freeze* en place sur la division, celle ci comportant des variables inconnues dans l'expression qui en constitue le diviseur. La multiplication, linéaire à l'appel, a pu être ajoutée au système de contraintes.

On pourra également noter ici les limitations de ce processus de retardement qui ne garantit en rien la détection systématique des ensembles de contraintes non-linéaires incohérents.

En voici deux exemples :

```
> { X*X = -2 };
  { X*X = -2 }
> { X*Y > 0, X > 0 > Y };
  { Y = -Y',
    (-Y')*X = U',
    X > 0,
    Y' > 0,
    U' > 0 }
>
```

ststs

Association Prolog
HERITAGE

Le contrôle et l'environnement des programmes

1. **Le contrôle**
2. **Expressions, variables statiques, tableaux**
3. **Structuration, saisie**
4. **Entrées / sorties**
5. **Autres éléments de l'environnement**

Que trouve-t-on dans ce chapitre ?

De tout…! Ce chapitre présente en effet un ensemble fort disparate de notions, qu'on pourrait regrouper sous l'appellation « les moyens de programmer effectivement en Prolog III ». Il y sera question des outils pour contrôler l'exécution des programmes afin de les rendre plus rapides, comme le célèbre /, ainsi que de la manière de structurer les gros programmes en modules et les directives pour interfacer harmonieusement ces derniers. On parlera aussi de l'ensemble des moyens réalisant pratiquement l'insertion, la suppression et la modification des règles, ainsi que leur sauvegarde et restauration à l'aide de fichiers magnétiques. Du coup, on en profitera pour traiter des entrées/sorties et, finalement de tous les autres points ayant trait au monde environnant le système Prolog III.

Décembre 1990

© PrologIA

1. Le contrôle

A chaque étape de l'exécution d'un programme l'interprète Prolog III doit faire deux choix : l'un pour choisir un but dans une suite de buts à exécuter, l'autre pour choisir la règle qui servira à l'exécuter. Au chapitre « Concepts de base » nous avons donné un modèle de l'exécution d'un programme Prolog III (voir le chapitre « Concepts de base ») :

(1) $(W, t_0\ t_1 \ldots t_n , S)$

(2) $s_0 \rightarrow s_1 \ldots s_m , R$

(3) $(W, s_1 \ldots s_m\ t_1 \ldots t_n , S \cup R \cup \{ s_0 = t_0 \})$

et nous avons expliqué comment se font les deux choix mentionnés : le but choisi est toujours le premier élément de la suite des buts à exécuter (ici t_0) ; les règles choisies pour l'exécuter sont toutes les règles dont "la tête s'unifie avec le but en question", c'est-à-dire les règles $s_0 \rightarrow s_1 \ldots s_m , R$ pour lesquelles le système $S \cup R \cup \{ s_0 = t_0 \}$ est soluble ; toutes ces règles sont examinées, *dans l'ordre où elles ont été écrites*.

Prolog III étant un langage foncièrement déclaratif, la notion de contrôle est nécessairement réduite à sa plus simple expression : cela consiste à modifier ou à restreindre les deux choix précédents. La manière dont Prolog fait ces choix peut amener certains programmes à boucler et par conséquent à ne pas se comporter comme on pouvait l'espérer. Les deux exemples suivants illustrent ce phénomène.

Exemple 1 : Un cas typique est celui de la transitivité. Quand on cherche à effacer *plus_grand(Jo,x)* en utilisant le programme suivant, on retrouve une instance de ce même but à effacer. Le programme se met alors à boucler et se termine par un débordement de pile (ou par une interruption utilisateur).

```
> insert;
plus_grand(Jo,Max) -> ;
plus_grand(Max,Fred) -> ;
plus_grand(x,y) -> plus_grand(x,z) plus_grand(z,y);;
{}
> plus_grand(Jo,x);
{x = Max}
{x = Fred}
```


© PrologIA

```
DEBORDEMENT
>
```

La manière correcte d'écrire ce programme consiste à enlever la récursivité à gauche :

```
> insert;
plus_grand'(Jo,Max)  ->;
plus_grand'(Max,Fred)  ->;

plus_grand(x,z)  ->
       plus_grand'(x,y)  plus_grand_ou_egal(y,z);

plus_grand_ou_egal(x,x)  ->;
plus_grand_ou_egal(x,y)  ->  plus_grand(x,y);;
{}
> plus_grand(Jo,x);
{x = Max}
{x = Fred}
>
```

Exemple 2 : Cet exemple énumère toutes les listes construites avec 1. Avec le (mauvais) programme ci-dessous, c'est d'abord la liste infinie qui devrait être produite :

```
> insert;
liste_de_un(<1>.x)  ->  liste_de_un(x);
liste_de_un(<>) ->;;
{}
> liste_de_un(x);

DEBORDEMENT
>
```

Bien entendu, la bonne solution s'obtient en permutant l'ordre des deux règles :

```
>  insert;
liste_de_un(<>)  ->;
liste_de_un(<1>.x)  ->  liste_de_un(x);;
{}
> liste_de_un(x);
{x = <>}
```

```
{x = <1>}
{x = <1,1>}
{x = <1,1,1>}

INTERRUPTION UTILISATEUR
```

La coupure « / » (cut)

Normalement, Prolog III essaye d'exécuter une suite de buts de toutes les manières possibles. Mais si on utilise une règle contenant un / (ou coupure[1]) pour exécuter un but *q*, l'exécution de ce / supprimera tous les choix de règles restant à faire pour l'exécution de *q*. Cela restreint la taille de l'espace de recherche : on peut dire que / fait «oublier» les autres manières possibles d'exécuter *q*.

Le «/» est un parasite qui ne peut apparaître que parmi les termes qui constituent le membre droit d'une règle. Les choix qui restent à examiner et que l'effacement du «/» supprime sont :

- les autres règles ayant la même tête que celle où le / figure
- les autres règles qui auraient pu être utilisées pour effacer les termes compris entre le début de la queue et le /

Cette question est illustrée par les exemples suivants :

```
> list;
couleur(rouge) ->;
couleur(bleu) ->;

taille(grand) ->;
taille(petit) ->;
```

1 En accord avec la syntaxe d'Edimbourg, la coupure peut se noter aussi « ! »

© PrologIA

```
choix1([x,y]) -> couleur(x) taille(y);
choix1("c'est tout") ->;

choix2([x,y]) -> / couleur(x) taille(y);
choix2("c'est tout") ->;

choix3([x,y]) -> couleur(x) / taille(y);
choix3("c'est tout") ->;

choix4([x,y]) -> couleur(x) taille(y) /;
choix4("c'est tout") ->;

{}
> choix1(u);
{u=[rouge,grand]}
{u=[rouge,petit]}
{u=[bleu,grand]}
{u=[bleu,petit]}
{u="c'est tout"}
> choix2(u);
{u=[rouge,grand]}
{u=[rouge,petit]}
{u=[bleu,grand]}
{u=[bleu,petit]}
> choix3(u);
{u=[rouge,grand]}
{u=[rouge,petit]}
> choix4(u);
{u=[rouge,grand]}
> choix1(u) /;
{u=[rouge,grand]}
>
```

On peut considérer la coupure comme une annotation que l'on fait à un programme *juste* pour le rendre plus *efficace* ; bien entendu, cela ne se justifie que si on ne s'intéresse qu'à la première solution fournie par ce programme.

Voici quelques utilisations classiques du «/» :

" Première solution uniquement "
premiere_solution_uniquement(b) -> b /;

" Si alors sinon "
si_alors_sinon(p,a,b) -> p / a;
si_alors_sinon(p,a,b) -> b;

```
" non "
    non(p) -> p / fail;
    non(p) ->;
```

Dans le cas de *non*, montré ci-dessus, il faut remarquer que l'on peut avoir des résultats inattendus si *p* contient des variables libres. C'est ce que montre le petit exemple suivant :

```
> list;
homme(Abelard) -> ;
femme(x) -> non(homme(x));
> femme(Abelard);
> femme(Eloise);
{}
> femme(x) eq(x,Eloise);
>
```

block(*e*, *b*), **block_exit(*e*)**
block est une règle prédéfinie qui permet de terminer brutalement l'exécution d'un but *b*. On peut considérer que :

- pour exécuter *block(e, b)* on exécute *b* en ayant auparavant créé une paire de parenthèses fictives, étiquetées par *e*, autour du but *b*.
- *block_exit(e)* provoque l'abandon immédiat de l'exécution de tous les buts inclus entre les parenthèses étiquetées par *e*. L'exécution continue ensuite normalement, après avoir restauré l'environnement comme il l'était avant l'exécution de *b*.

De plus :

- l'étiquette dont il est question ci-dessus est un terme Prolog III quelconque, et on considère que pour que deux termes représentent la même étiquette au regard du mécanisme *block_exit*, il faut et il suffit qu'ils soient unifiables.
- s'il y a plusieurs parenthèses étiquetées par *e*, *block_exit(e)* s'arrête au couple de parenthèses la plus interne.
- si *block_exit(e)* ne rencontre pas de parenthèses étiquetées par *e*, alors une erreur est signalée.

© PrologIA

Interruption
A tout instant un programme Prolog peut être interrompu au moyen d'une touche déterminée, dépendant du système utilisé (par exemple : <Ctrl-C>). Un petit dialogue est alors proposé à l'utilisateur :

User Interrupt by (^C). C(ontinue, K(ill, T(race, Q(uit ?

lui permettant selon la réponse fournie, de continuer l'exécution *(C)*, de l'interrompre *(K)*, de continuer l'exécution en mode trace *(T)* ou encore de quitter la session *(Q)*.

Les primitives suivantes sont considérées aussi comme des moyens de contrôler l'exécution d'un programme :

bound(*x*)
bound(x) s'efface si *x* est lié. Une variable est considérée comme liée si elle est astreinte à représenter un arbre dont

- l'étiquette initiale est connue
- le nombre de fils est connu

default(t_1, t_2)
La règle prédéfinie *default* permet de réaliser le contrôle suivant : si on peut effacer t_1, alors on l'efface *de toutes les manières possibles*, sinon on efface t_2, également de toutes les manières possibles. Il faut remarquer que cette primitive ne peut pas être réalisée avec « / ». Voyons un exemple d'utilisation de cette règle :

```
> list;
repondre(p) -> default(p,outml("personne"));

homme(jean) ->;
homme(pierre) ->;

{}
> repondre(homme(x));
{x=jean}
{x=pierre}
> repondre(femme(x));
personne
{}
>
```

free(*x*)
S'efface uniquement si *x* n'est pas, lié c'est-à-dire si l'on ne connait pas l'étiquette de l'arbre représenté par *x* et si l'on ne sait pas si le nombre de fils de *x* est nul ou pas.

2. Expressions, variables statiques, tableaux

Expressions arithmétiques

En Prolog III les opérations arithmétiques sont intégrées dans le cœur même du langage. Cependant, nous avons conservé le mécanisme *val* de Prolog II, par souci de compatibilité, parce que ce mécanisme reste dans certaines circonstances un procédé fort efficace pour évaluer des expressions arithmétiques, et surtout parce qu'il demeure le moyen d'utiliser les variables statiques et les tableaux.

Le résultat d'une évaluation faite au moyen de la règle prédéfinie *val* est soit un entier, soit un réel, soit un identificateur, soit une chaîne. Les valeurs booléens, dans ce cas seront représentées par les entiers 0 et 1. Sur la plupart des machines, les valeurs des entiers sont comprises entre -2 147 483 648 et 2 147 483 647 (2^{31}-1) ; celles des flottants entre -1.7e38 et +1.7e38 et la plus grande valeur négative et la plus petite valeur positive sont respectivement -0.29e-38 et +0.29e-38. Les valeurs situées strictement entre ces deux valeurs sont mises à 0.

val(t_1, t_2)

Evalue l'expression t_1 et produit le résultat t_2. L'expression à évaluer est construite récursivement à partir de constantes, de variables statiques (identificateurs assignés), d'identificateurs, de composantes de tableaux et de fonctions évaluables. Des variables Prolog III ordinaires peuvent apparaître dans une expression évaluable, à la condition qu'elles soient contraintes à représenter une valeur connue.

© PrologIA

Exemples:

```
> val(add(mul(2,add(3,4)),1000),x);
{x = 1014}
> val(2*(3+4)+1000,x);
{x = 1014}
>
```

L'intégration de l'arithmétique au cœur de Prolog III fait que les expressions évaluables n'ont souvent pas la même signification qu'en Prolog II, du moins lorsqu'elles sont écrites dans la bonne syntaxe. Ecrivons les deux termes qui ont été évalués dans l'exemple précédent :

```
> {t=add(mul(2,add(3,4)),1000)};
{t = add(mul(2,add(3,4)),1000)}
> {t=2*(3+4)+1000};
{t = 1014}
>
```

Lorsqu'une fonction à évaluer a un nombre incorrect d'arguments ou lorsque certains de ses arguments sont du mauvais type, il se produit une erreur. Chaque fonction évaluable doit avoir des arguments d'un type précis : aucune conversion automatique de type n'est effectuée lors de l'évaluation.

- La valeur d'un nombre ou d'une chaîne est égale à ce nombre ou à cette chaîne.
- La valeur d'un tableau indicé est égale à la valeur de l'élément correspondant de ce tableau. Exemple:

```
> def_array(tab,100) assign(tab(50),3);
{}
> val(tab(50),x);
{x = 3}
>
```

- La valeur d'un identificateur *i* est définie comme suit :
  - si une constante *k* a été affectée à *i* (au moyen de la règle *assign*, voir le paragraphe suivant) alors la valeur de *i* est *k*.
  - si *i* n'a pas fait l'objet d'une affectation préalable, alors la valeur de *i* est *i* lui-même.

Exemple :

```
> assign(un,1);
{}
> val(un,x) val(deux,y);
{x = 1, y = deux}
>
```

Les fonctions évaluables sont :

add(t_1, t_2)
valeur(add(t_1, t_2)) = valeur(t_1) + valeur(t_2).

Les valeurs de t_1 et t_2 doivent être numériques et connues.

sub(t_1, t_2)
valeur(sub(t_1, t_2)) = valeur(t_1) - valeur(t_2).

Les valeurs de t_1 et t_2 doivent être numériques et connues.

mul(t_1, t_2)
valeur(mul(t_1, t_2)) = valeur(t_1) × valeur(t_2).

Les valeurs de t_1 et t_2 doivent être numériques et connues.

div(t_1, t_2)
$$valeur(div(t_1, t_2)) = \frac{|\, valeur(t_1)\,|}{|\, valeur(t_2)\,|}$$

Il s'agit du quotient entier par défaut de la valeur absolue de t_1 par la valeur absolue de t_2. Les valeurs de t_1 et t_2 doivent être des nombres entiers.

mod(t_1, t_2)
valeur(mod(t_1, t_2)) = |valeur(t_1)| modulo |valeur(t_2)|.

Il s'agit du reste de la division entière de la valeur absolue de t_1 par la valeur absolue de t_2. Les valeurs de t_1 et t_2 doivent être des nombres entiers.

© PrologIA

eql(t_1, t_2)
valeur(eql(t_1, t_2)) = 1 si valeur(t_1) = valeur(t_2) , 0 sinon.

Les valeurs de t_1 et t_2 doivent être des constantes de même type.

inf(t_1, t_2)
valeur(inf(t_1, t_2)) = 1 si valeur(t_1) < valeur (t_2) , 0 sinon.

Les valeurs de t_1 et t_2 doivent être des constantes de même type. Pour les entiers et les réels, on prend la relation "<" entre les nombres. Pour les chaînes, on prend l'ordre lexicographique et pour les identificateurs on prend l'ordre lexicographique sur les chaînes associées.

infe(t_1, t_2)
valeur(infe(t_1, t_2)) = 1 si valeur(t_1) ≤ valeur (t_2) , 0 sinon.

Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

sup(t_1, t_2)
valeur(sup(t_1, t_2)) = 1 si valeur(t_1) > valeur (t_2) , 0 sinon.

Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

supe(t_1, t_2)
valeur(supe(t_1, t_2)) = 1 si valeur(t_1) ≥ valeur (t_2) , 0 sinon.

Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

if(t, t_1, t_2)
valeur(if(t, t_1, t_2)) = valeur(t_1) si (valeur(t)≠0) , valeur(t_2) sinon.

trunc(t)
valeur(trunc(t)) = conversion en entier de la valeur de t.

abs(t)
valeur(abs(t)) = valeur absolue de valeur(t).

La valeur de *t* doit être de type entier ou flottant.

Les fonctions suivantes donnent un résultat de type réel. Les fonctions trigonométriques travaillent avec des angles exprimés en radians.

atan(*t*)
valeur (atan(t)) = arc tangente de valeur(t).

cos(*t*)
valeur (cos(t)) = cosinus(valeur(t)).

exp(*t*)
valeur (exp(t)) = exponentielle(valeur(t)).

ln(*t*)
valeur(ln(t)) = logarithme népérien(valeur(t)).

rad(*t*)
valeur (rad(t)) = conversion en radian(valeur(t)).

sin(*t*)
valeur(sin(t)) = sinus(valeur(t)).

sqrt(*t*)
valeur(sqrt(t)) = racine carrée(valeur(t)).

tan(*t*)
valeur(tan(t)) = tangente(valeur(t)).

Affectation. Tableaux.

assign(*i*, *t*)
Affecte le terme *t*, qui doit représenter une constante, à l'identificateur *i*. Tout se passe comme si *i* devenait le nom d'une variable «globale» (ultérieurement accessible pendant l'exécution de n'importe quel but) et «statique» (résistante au backtracking) possédant *t* pour valeur. Il s'agit donc bien de l'affectation classique, comme elle se pratique en FORTRAN, Pascal, etc….

© PrologIA

Exemple :

```
> assign(nom_fichier,"monfichier.txt");
{}
> val(nom_fichier,x);
{x="monfichier.txt"}
>
```

En Prolog, ces variables statiques peuvent être vues comme une manière particulièrement efficace d'écrire des assertions (ou règles sans queue). Du point de vue de l'utilisateur, on peut considérer que l'emploi de *assign* et *val* tel qu'il est fait ci-dessus est équivalent à :

```
> retract(nom_fichier(x),[]);
> assert(nom_fichier("monfichier.txt"),[]);
{}
> nom_fichier(x);
{x="monfichier.txt"}
>
```

assign(*tab*(*i*), *t*)
Affecte la valeur de *t* à l'élément de rang *i* du tableau *tab*. Ce tableau doit avoir été défini au préalable avec *def_array* (voir ci-dessous). Les mêmes restrictions que précédemment s'appliquent à *t* : ce doit être une constante. Lorsqu'on veut obtenir la valeur d'un élément de rang *i* du tableau, si cet élément n'a pas été affecté (*assign*) ou encore si la valeur du rang *i* est à l'extérieur des bornes du tableau *val* produit une erreur.

Exemple :

```
> def_array(tableau,10);
{}
> assign(tableau(5),  cinq);
{}
> val(tableau(5),  x);
{x = cinq}
> val(tableau(4),  x);
Erreur 246: Erreur dans val
> val(tableau(11),  x);
Erreur 246: Erreur dans val
>
```

def_array(*i*, *n*)
Définit dynamiquement un tableau de constantes Prolog de nom *i* et de taille *n*. Ce tableau se comportera comme une variable globale et statique. Les valeurs légales de l'indice sont les éléments 1, 2, … n.

Si un tableau de même nom existe déjà :
- s'il s'agit d'un tableau de même taille, il ne se passe rien
- si les tailles diffèrent, il se produit une erreur

L'accès et l'affectation aux composantes des tableaux sont analogues à ceux des tableaux des autres langages de programmation. Le tableau est désalloué lorsqu'on supprime (par exemple par *kill_module*) le module auquel il appartient.

Exemple : la gestion d'une pile.

```
> insert;
inc(i) -> val(i,x) assign(i,x+1);
dec(i) -> val(i,x) assign(i,x-1);

initialise -> assign(pointeur,1) def_array(pile,100);

empile(v) ->
     val(pointeur,p)  !
     assign(pile(p), v)
     inc(pointeur),
     { p <= 100 };
empile(v) -> outml("débordement de la pile") fail;

depile(v) ->
     dec(pointeur)
     val(pointeur,p)  !
     val(pile(p),v)
     dec(pointeur),
     { p >= 1 };
depile(v) -> outml("pile vide") assign(pointeur,1) fail;;
{}
> initialise;
{}
> empile(111);
{}
> empile(222);
{}
> depile(x)  depile(y);
```


© PrologIA

```
{x=222, y=111}
> depile(x);
pile vide
>
```

3. Structuration, saisie et modification des règles

Tous les exemples que nous avons examiné jusqu'à présent étaient de petits programmes Prolog III dans lesquels le nombre des identificateurs restait très limité et facile à appréhender par le programmeur. Dans de telles circonstances, on peut supposer que les identificateurs en présence appartiennent à un unique "univers" qui les contient tous ; c'est ce que nous avons fait, faute d'indication contraire. Tant que les programmes restent de taille raisonnable, la nécessité de regrouper les identificateurs et les règles dans des ensembles fonctionnels ne se manifeste pas.

La situation n'est plus la même dès qu'on considère de gros programmes suffisamment complexes pour nécessiter une décomposition en modules de fonctionnalités différentes ou encore des programmes, même relativement simples, qui comportent un nombre important d'identificateurs, qu'ils soient ou non des noms de règles. Apparaît alors très vite l'utilité de disposer d'un moyen pour regrouper les identificateurs selon des critères "logiques", ces regroupements étant un moyen pour manipuler aisément des familles entières d'identificateurs à l'aide de noms uniques, et en même temps un moyen pour réglementer leur visibilité, c'est-à-dire pour indiquer quels noms définis dans une "partie d'un programme" (nous dirons *module*) peuvent être référencés dans les autres parties.

Il s'agit donc de partitionner l'ensemble des identificateurs connus par un programme Prolog III. En peu de mots, les outils pour effectuer ce partitionnement peuvent se résumer de la manière suivante :

- la syntaxe complète des identificateurs prescrit la présence d'un *préfixe* au début de tout identificateur

- l'ensemble des identificateurs ayant même préfixe est une *famille*. Les familles peuvent être manipulées globalement : de nombreuses primitives agissent sur des familles entières, représentées par le préfixe commun de leurs membres.
- certaines conventions, définissant des *contextes d'écriture/lecture*, permettent d'écrire les identificateurs en omettant leur préfixe et inversement de compléter des identificateurs lus sans préfixe.

Voyons tout cela un peu plus en détail.

Familles d'identificateurs

Voici la syntaxe complète des identificateurs :

<préfixe>
::= ⟦ <nom> { **:** <nom> } ⟧ *orbis:lex*

<nom>
::= <lettre> { <alphanumérique étendu> }

<identificateur>
::= <préfixe> **:** <identificateur abrégé> *sys:outm*
::= <identificateur abrégé> *outm*

<identificateur abrégé>
::= <lettre> <lettre> { <alphanumérique étendu> }

Un *identificateur complet* se compose donc d'un nom précédé d'un *préfixe*, dont il est séparé par le caractère deux-points. Un identificateur abrégé est ce qui reste d'un identificateur complet lorsqu'on omet le préfixe et le séparateur. Le préfixe peut être vide, ou bien être un mot, ou encore être composé de plusieurs mots séparés par des deux-points.

© PrologIA

Un identificateur complet comporte toujours au moins une occurrence du caractère « : » tandis qu'un identificateur abrégé n'en comporte aucune. Par exemple, les trois premiers identificateurs ci-dessous sont complets, le dernier est abrégé :

```
grammaire:pluriel
sys:env:screen:clear_screen
:paul
paul
```

Les préfixes respectifs de ces trois premiers exemples sont **"grammaire"**, **"sys:env:screen"** et **""** (le préfixe vide). Les identificateurs abrégés associés sont respectivement **pluriel**, **clear_screen** et **paul**.

On appelle *famille* correspondant à un préfixe donné l'ensemble des identificateurs ayant ce préfixe. Il n'y a pas de hiérarchie dans les familles : lorsqu'un identificateur complet comporte plusieurs occurrences du caractère « : » seule la dernière est prise en considération par Prolog III, comme séparateur du préfixe et de l'identificateur abrégé. Le préfixe est alors composé de plusieurs mots séparés par des « : ». Si cela correspond à une organisation hiérarchique dans l'esprit du programmeur, ce fait reste étranger à Prolog III, pour qui les familles associées à des préfixes comme par exemple **"aa"** et **"aa:bb"** n'ont pas de lien particulier.

On appelle *module* l'ensemble des éléments de code nommés par les identificateurs d'une famille donnée. Par « éléments de code » nous entendons bien sûr les faits et les règles Prolog III qui constituent les programmes, mais aussi les fonctions évaluables, les variables statiques[1] et les tableaux.

1 C'est-à-dire les identificateurs auxquels on associe une valeur par la primitive *assign*.

Partie fermée d'une famille

Les notions de famille et de module sont tout-à-fait dynamiques : chaque nouvel identificateur lu et éventuellement complété par Prolog III est incorporé automatiquement à la famille correspondant à son préfixe. Une famille n'est jamais figée, elle peut à tout moment être augmentée par l'ajout de nouveaux identificateurs. Ce dynamisme est en général fort appréciable, mais nous allons voir que lors de la définition des contextes de lecture et écriture (voir ci-dessous) une certaine stabilité est aussi souhaitable, afin que les conventions définies par un contexte soient autant que possible "hors du temps" et qu'un identificateur abrégé représente le même identificateur complet à tout moment d'une session Prolog III, ainsi que d'une session à une autre.

Ceci justifie l'introduction de la notion de famille fermée, ou plutôt de *partie fermée d'une famille* : des primitives sont fournies (*close_context_dictionary*) pour figer un « noyau dur » de la famille, égal à l'état de cette dernière au moment ou la fermeture est effectuée. Par la suite, d'autres identificateurs pourront encore être ajoutés à la famille, mais ils ne seront pas incorporés à la partie fermée de cette dernière. Ce concept sera exploité dans l'interprétation de la *liste implicite* qui figure dans les primitives de définition des contextes de lecture et écriture : à cet endroit, une référence à une famille ne désigne que sa partie fermée. Une telle référence est donc stable.

Contexte de lecture et d'écriture

Un *contexte de lecture et écriture* est un ensemble de règles de transformation d'identificateurs abrégés en identificateurs complets et vice-versa. Elles associent à tout identificateur abrégé un unique identificateur complet ; inversement, elles définissent l'ensemble des identificateurs qui peuvent s'écrire sous forme abrégée. Le codage interne des identificateurs met toujours en œuvre la forme complète de ceux-ci ; les transformations dont nous parlons sont donc faites uniquement au moment de leur lecture ou de leur écriture.

© PrologIA

Un contexte de lecture et écriture par défaut est défini au démarrage de Prolog III et des primitives sont fournies qui vous permettent de définir vos propres contextes, de les nommer, de les ranger pour les rappeler ultérieurement, etc… Un (et un seul) contexte de lecture et écriture est constamment défini tout au long d'une session Prolog III ; nous l'appellerons le *contexte courant*.

Formellement, un contexte de lecture et écriture est défini par la donnée d'un triplet

(*liste explicite* , *liste implicite* , *préfixe par défaut*)

Où *liste explicite* et *liste implicite* désignent, par un mécanisme expliqué plus loin, une grande liste *L* d'identificateurs complets. Les conventions définies par un tel contexte sont les suivantes : lors d'une lecture

- un identificateur complet n'est nullement transformé
- un identificateur abrégé coincidant avec la forme abrégée d'un élément *i* de *L* est considéré comme représentant cet identificateur, et est donc complété avec le préfixe de *i*
- les autres identificateurs abrégés sont complétés avec le *préfixe par défaut* indiqué.

Par exemple, si la liste *L* est *[aa:bb, aa:cc, aa:dd]* et le préfixe par défaut *"zz"*, alors

- *pp:xx* est compris comme *pp:xx*, car lu sous sa forme complète
- *cc* est compris comme *aa:cc*, puisque ce dernier figure dans *L*
- *xx* est compris comme *zz:xx*, puisqu'aucun identificateur complet de *L* n'a *xx* pour forme abrégée.

Venons-en aux détails pratiques de la définition des contextes de lecture et écriture dans les programmes Prolog III.

set_context(*nom*, *suite_explicite*, *suite_implicite*, *préfixe_par_défaut*)
Cette commande permet à la fois de définir un contexte, en lui associant un nom, et de l'installer comme contexte courant de lecture et écriture.

- *nom* est un identificateur ou une chaîne : il sert de nom au contexte nouvellement défini.
- *suite_explicite* est une liste de la forme

  [$prefixe_1$, $liste_1$, $prefixe_2$, $liste_2$,…, $prefixe_n$, $liste_n$]

  qui représente une suite d'identificateurs complets, « factorisés » par familles. Par exemple, la liste :

  ["m1",["aa","bb"],"m2",["cc"],"m3",["dd","ee","ff"]]

  représente en fait la liste d'identificateurs complets *m1:aa*, *m1:bb*, *m2:cc*, *m3:dd*, *m3:ee*, *m3:ff*
- *liste_implicite* est une liste de préfixes donnés sous forme de chaînes de caractères. Chacun de ces préfixes représente la partie fermée de la famille correspondante ou, si la fermeture n'a pas été réalisée, la totalité de cette famille.

  Dans ces conditions, la liste *L* d'identificateurs définie par tous ces éléments est la concaténation, dans l'ordre où elles apparaissent, de la *suite_explicite* et des listes d'identificateurs définies par la *suite_implicite*. L'exploitation ultérieure de toute cette information respecte l'ordre d'entrée en scène de ces éléments : les recherches se font d'abord dans la liste déduite de *suite_explicite*, cette liste étant parcourue du début vers la fin, puis dans la liste déduite du premier élément de *suite_implicite*, puis dans celle déduite de son second élément, et ainsi de suite.
- *préfixe_par_défaut* est le préfixe qui sera attribué à chaque identificateur abrégé à condition que ce dernier ne soit pas la forme abrégée de l'un des identificateurs apparaissant dans la liste *L*.

Note 1. L'exécution de la commande *set_context(nom, suite_explicite, suite_implicite, préfixe_par_défaut)* produit la mémorisation du contexte décrit par cette commande, sous la forme d'un fait ajouté au module d'environnement :

sys:env:context(nom, suite_explicite, suite_implicite,

© PrologIA

préfixe_par_défaut) -> ;

Note 2. Si vous définissez des contextes ne contenant pas la famille *"sys"* dans *suite_implicite*, vous devrez faire attention au fait qu'un grand nombre d'identificateurs abrégés perdent leur signification habituelle : des règles prédéfinies deviennent inconnues sous leur nom abrégé et, piège bien plus redoutable, des identificateurs abrégés très communs, comme *[]* (l'équivalent du *nil* de Prolog II) n'ont plus la forme complète qui est utilisée dans les règles prédéfinies qui constituent la bibliothèque-système de Prolog III.

Note 3. Lorsqu'une règle contient une déclaration de contexte, celle-ci ne devient effective que lorsque la règle est exécutée (et non au moment de sa lecture). Dans l'exemple suivant, l'identificateur abrégé *peche* est complété en *def:peche* et non en *fruits:peche*, puisque la définition du contexte *"new"* ne sera effective que lors de l'exécution du programme *pomme* :

```
> pomme  ->
    set_context("new",["fruits",["peche"]],["sys"],
        "def")
    peche;
```

set_context(*nom*)
Permet de réinstaller comme contexte courant un contexte précédemment défini. L'argument nom doit être l'identificateur ou la chaîne associée au contexte en question lors de sa définition.

current_context(t_0, t_1, t_2, t_3)
current_context(t_0)
Obtention du nom et de la définition du contexte courant :

t_0 = *nom* du contexte courant (identificateur ou chaîne) ;
t_1 = *suite_explicite*,
t_2 = *suite_implicite*,
t_3 = *préfixe_par_défaut*.

close_context_dictionary(*s*)
Définit l'ensemble actuel d'identificateurs de la famille de préfixe *s* comme étant la *partie fermée* de cette famille. Les identificateurs ajoutés à la famille après l'exécution de cette commande ne seront pas automatiquement incorporés à cette partie fermée.

add_implicit(s_1, s_2)
remove_implicit(s_1, s_2)
Ajoute à (resp. enlève de) la partie fermée de la famille correspondant au préfixe s_1 l'identificateur dont le nom abrégé est donné par la chaîne s_2. Examinons l'effet de ces commandes sur un exemple :

```
> current_context(t0,t1,t2,t3);
{ t0 = 1, t1 = '[]', t2 = ["sys"], t3 = <> }
> string_ident(p,a,dictionary)  outl(sys:dictionary);
dictionary
{p = "sys", a = "dictionary"}
>
```

en lecture, l'identificateur abrégé *dictionary* est devenu *sys:dictionary* (comme le montre le résultat du prédicat prédéfini *string_ident*) tandis qu'en écriture, l'identificateur *sys:dictionary* est abrégé en *dictionary*. Vue la définition du contexte, tout ceci montre que *sys:dictionnary* appartient à la partie fermée de la famille *"sys"*. Continuons notre exemple :

```
> remove_implicit("sys","dictionary");
{}
> string_ident(p,a,dictionary)  outl(sys:dictionary);
sys:dictionary
{p = <>, a = "dictionary"}
>
```

notre identificateur *dictionary* n'appartient plus à la partie fermée de la famille *"sys"*. Le comportement du contexte de lecture et écriture à son sujet a changé : en lecture, *dictionary* est complété en *:dictionary*, puisque d'après le contexte le préfixe par défaut est la chaîne vide (que l'on peut noter *""* ou <>) tandis que en écriture, *sys:dictionary* n'est pas transformé en sa forme abrégée.

dictionary

© PrologIA

Ecrit sur la sortie courante la forme abrégée des *accès* qui ont le même préfixe que le préfixe par défaut du contexte de lecture et écriture. Par accès on entend un identificateur qui nomme un objet dans le code : règle, fait, variable statique ou tableau.

dictionary(L)
Unifie *L* avec la liste des préfixes des modules utilisateurs présents en mémoire.

dictionary(s_0, t_0)
s_0 étant une chaîne de caractères, cette primitive unifie t_0 avec la liste des règles de la famille correspondant au préfixe s_0. Chaque règle est représentée par un couple < *accès* , *arité* >

Modules

Dans un programme donné, et à un moment donné, on appelle *module p* le regroupement de tous les faits et règles, les fonctions évaluables, les variables statiques (ou identificateurs affectés par *assign*) et les tableaux dont le nom appartient à la même famille *p*, c'est-à-dire possède le même préfixe *p*, que nous appellerons le *préfixe du module*. Les identificateurs de la famille *p* peuvent apparaître dans d'autres modules, mais pas en position d'identificateur d'accès (c'est-à-dire comme nom de prédicat de tête d'une règle). La notion de module est dynamique : lorsqu'une nouvelle règle est créée ou supprimée, le module correspondant à son identificateur d'accès est modifié en conséquence.

Un module peut exister sous forme de texte (avant lecture) ou sous forme interne (après lecture), on parlera de module source et de module objet pour désigner l'un et l'autre cas.

Module source. Un *module source* est défini comme la suite des règles comprises entre les deux directives:

```
module(préfixe-du-module,
                suite-explicite, suite-implicite, préfixe-par-défaut);
....
```

end_module(*préfixe-du-module*);

La directive *module* fait basculer Prolog III dans le mode insertion et la directive *end_module* le fait sortir de cet état. Par conséquent, les éléments compris entre ces deux directives doivent être des règles (ou des commentaires, qui sont ignorés).

L'argument désignant le *préfixe-du-module* dans la directive *module* doit être le même que celui figurant dans la directive *end_module*, et toutes les règles comprises entre ces deux directives doivent avoir un identificateur d'accès avec ce préfixe ; ceci permet de détecter à la lecture les éventuelles erreurs de nommage. Naïvement, on peut dire que la directive *module* signifie « *insérez les règles qui suivent, avec le contexte de lecture défini ici, et en vérifiant que les règles lues possèdent (ou acquièrent) bien le préfixe voulu* »

Lorsque tous les arguments ne sont pas donnés, on a les valeurs par défaut suivantes (en supposant que le contexte lors de la lecture de la directive *module* dispense de l'écrire *sys:module*):

```
module(p)       ⇔    module(p,  [],  ["sys"],  p)
module(p,s)     ⇔    module(p,   s,  ["sys"],  p)
module(p,s,i)   ⇔    module(p,   s,     i,     p)
```

Le contexte de lecture défini par l'en-tête du module est utilisé pour la lecture de tout le reste du module, y compris la directive *end_module*. Le contexte qui existait avant la lecture du module est ensuite rétabli.

Si un module *m* contient une règle avec le préfixe *m* et le nom abrégé *ini_module* alors celle-ci est automatiquement exécutée après la lecture (ou le chargement ultérieur) du module.

Le contexte de lecture du module permet de redéfinir de manière claire certaines des primitives utilisées à l'intérieur d'un module.

Exemple :

© PrologIA

```
module("exemple", ["exemple", ["out"]] );
...
out(x) -> enjoliver(x,y) sys:out(y);
...
end_module("exemple");
```

Module objet. Une fois lus par Prolog III, les éléments de code qui forment un module sont dits constituer un *module objet*. La manipulation de modules objets ne concerne que les représentations internes des identificateurs, la notion de contexte n'intervient donc pas dans ces opérations.

Un module objet peut être sauvegardé dans un fichier magnétique en vue de son rechargement ultérieur. Cela se fait par les primitives *load* et *save*. Lorsqu'un module objet est écrit dans un fichier, il est rangé de telle manière qu'il est possible, lors de la restauration du module, de renommer ses identificateurs. Il est ainsi possible de résoudre les conflits de noms quels qu'ils soient, et de construire des programmes qui n'interfèrent pas avec les données qu'ils manipulent.

Ajout, suppression et recherche de règles

assert(*t*, *q*)
asserta(*t*,*q*)
Ajout d'une règle, au début de son paquet.

t doit être un terme pouvant être une tête de règle, c'est-à-dire un terme qui représente un arbre dont le nœud initial comporte une étiquette connue, qui est un identificateur. *q* doit être une liste de termes. La règle effectivement ajoutée par ces primitives comporte un système vide de contraintes.

L'exécution de *assert(t, [q_1, … q_n])* ajoute la règle *t* -> q_1 … q_n , { } au-dessus du groupe de règles ayant le même «nom» que *t*, c'est-à-dire au début du paquet correspondant à *t*, et si aucun groupe de règles n'a le même «nom» que *t*, un nouveau paquet est créé. Par exemple, les deux commandes suivantes

```
assert(conc([e|x],y,[e|z]),[conc(x,y,z)]);
assert(conc([],y,y),[]);
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
conc([],y,y)  ->;
conc([e|x],y,[e|z])  ->  conc(x,y,z);
```

assert''(*t*, *q*)
assertz(*t*, *q*)
Ajout d'une règle, à la fin de son paquet.

Même fonctionnement que *assert*, mais l'ajout se fait au-dessous, et non au-dessus, du paquet correspondant à *t*. Exemple : les deux commandes suivantes, dans l'ordre indiqué, produisent l'insertion du même programme *conc* que ci-dessus :

```
assert''(conc([],y,y),[]);
assert''(conc([e|x],y,[e|z]),[conc(x,y,z)]);
```

current_predicate(<*i*,*a*>)
Permet de tester la présence d'une règle.

L'exécution de cette primitive réussit s'il existe une règle de nom *i* et d'arité *a*. Si *a* est une variable, et *i* est connu, elle énumère successivement toutes les arités de règles de nom *i*. Si *i* est une variable, et *a* est connu, elle énumère successivement tous les noms de règles d'arité *a*. Enfin si *a* et *i* sont des variables, cette primitive énumère successivement tous les couples <*i*,*a*> correspondant aux règles du module associé au préfixe par défaut du contexte courant.

insert
reinsert
Insertion de règles.

Cette règle prédéfinie fait basculer le système dans un mode dans lequel les énoncés (règles, commentaires et directives) lus sur l'unité d'entrée courante sont ajoutés, dans l'ordre dans lequel ils sont lus, au programme courant.

© PrologIA

Les directives (*set_context*, *module*, etc…) sont exécutées immédiatement lorsqu'elles sont rencontrées. Le mode *insert* se termine soit quand un énoncé vide est trouvé, soit si l'on rencontre la fin du fichier d'entrée.

Exemple:

```
> insert;
conc([],y,y)  ->;
conc([e|x],y,[e|z])  ->  conc(x,y,z);;
>
```

Si une erreur de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un « ; », ou jusqu'à la fin de la ligne sur l'unité *console*) sont ignorés.

insert provoque une erreur lorsque un paquet lu existe déjà, alors que *reinsert* remplace l'ancien paquet par la nouvelle définition. Attention ceci peut être dangereux : par exemple une erreur sur une règle à l'intérieur d'un paquet peut provoquer, avec *reinsert*, l'écrasement des règles précédentes quand on continuera à lire le paquet.

insert(*f*)
reinsert(*f*)
Insertion de règles à partir d'un fichier.

Même fonctionnement que *insert*, mais les énoncés sont lus sur le fichier indiqué. Selon que l'on a par ailleurs activé ou non le mode écho (cf. règle *echo*) les règles sont affichées sur la console au fur et à mesure de leur lecture.

list
Liste sur la sortie courante toutes les règles déterminées par le préfixe par défaut du contexte courant.

list(*t*)
Liste sur la sortie courante le ou les paquet de règles indiqués par *t*, qui doit être soit un terme de la forme indiquée ci-dessous, soit une liste de tels termes (*c* désigne une chaîne de caractères, *i* un identificateur, *a* un entier, et *v* une variable) :

c Toutes les règles du module de nom *c*

<i, a>	Toutes les règles composant le paquet de nom *i* et d'arité *a*.
<i, v>	Toutes les règles dont l'accès est *i*, quelque soit leur arité.
<v, a>	Toutes les règles d'arité *a* appartenant au module ayant le préfixe par défaut du contexte de lecture et écriture courant dont l'arité est *a*.
i	équivalent de *<i, v>*

Exemple :

```
> list([<aaa,2>,<aaa,3>,<bbb,X>]);
```

list(*<i,a>*, *n*)
Lister une règle.

i doit être un identificateur et *a* un entier; la règle numéro *n* du paquet correspondant au nom *i* et à l'arité (nombre d'arguments) *a* est listée sur la sortie courante.

predefined(*t*)
Cette primitive réussit si *t* est un terme correspondant à l'appel d'une règle prédéfinie.

rule(*n*, *a*, *t*, *q*)
rule(*n*, *t*, *q*)
Recherche des règles correspondant à un *modèle* donné.

Ces primitives énumèrent les règles qui satisfont aux contraintes suivantes : *n* = rang de la règle dans son paquet, *a* = identificateur d'accès (nom) de la règle, *t* = tête de la règle, *q* = queue de la règle. Dans la première forme, a doit être un identificateur connu ; dans la deuxième forme, *t* doit être suffisamment connu pour pouvoir en déduire l'identificateur d'accès des règles visées.

Exemple d'utilisation se rapportant toujours au programme *conc* donné plus haut en exemple :

```
> rule(n,conc,t,q);
```


© PrologIA

```
{ n = 1, t = conc([],x_8,x_8), q = [] }
{ n = 2, t = conc([e_11 | x_11],y_11,[e_11 | z_11]),
  q = [conc(x_11,y_11,z_11)] }
>  rule(n,conc[v],q);
{ n = 1, v = <[],x_12,x_12>, q = [] }
{ n = 2, v = <[e_15 | x_15],y_15,[e_15 | z_15]>,
  q = [conc(x_15,y_15,z_15)] }
>
```

retract(*t*, *q*)
Recherche et suppression des règles correspondant à un *modèle* donné.

Cette primitive recherche toutes les règles qui satisfont aux contraintes t = tête de la règle, q = queue de la règle, et les supprime. *t* doit être assez connu pour qu'il n'existe aucune ambiguïté quant au nom des règles visées.

suppress(*i*)
suppress(*i*, *a*)
suppress(*i*, *a*, *n*)
Suppression de règles.

i doit être un identificateur, *a* et *n* des entiers; ces primitives produisent la suppression de :
- première forme : toutes les règles ayant l'identificateur d'accès (nom) *i*
- deuxième forme : toutes les règles de nom *i* et d'arité *a*
- troisième forme : la *n*-ème règle parmi celles qui ont le nom *i* et l'arité *a*

Exemple :

```
> insert;
data(1)  ->;
data(2)  ->;
data(3)  ->;
> data(x);
{x = 1}
{x = 2}
{x = 3}
> suppress(data,1,2);
{}
> data(x);
{x = 1}
{x = 3}
>
```

Manipulation de modules objets

kill_module(*s*)
Supprime toutes les règles du module désigné par la chaîne de caractères *s*. Les tableaux du module sont désalloués, et les assignations des identificateurs du module sont défaites.
reload(*f*, *l*)
Chargement de modules sauvés.

f est un nom de fichier (chaîne de caractères) et *l* est une liste de substitution de préfixes. Cette commande produit le chargement des modules sauvegardés dans le fichier indiqué; celui-ci doit avoir été produit par la commande *save*. En cas de redéfinition d'un élément, la version rencontrée dans le fichier remplace celle qui se trouve en mémoire, sans produire d'erreur.

l est une liste de la forme [<$pref_1$, $subs_1$> , … <$pref_k$, $subs_k$>] qui spécifie le renommage des modules chargés : $pref_1$ sera substitué par $subs_1$, $pref_2$ par $subs_2$, etc… Cette liste peut être vide.

© PrologIA

save(*l*, *f*)
Sauvegarde de modules.

f est un nom de fichier (chaîne de caractères) et *l* une liste de préfixes (chaînes de caractères). Cette commande produit la sauvegarde dans le fichier indiqué de tous les éléments (règles, variables statiques et tableaux) des modules correspondant aux préfixes donnés.

Exemple:

```
> save(["","data","dict"],  "myfile.mo");
{}
```

Le fichier produit est un fichier de *code objet*, destiné à être ultérieurement chargé dans un autre environnement Prolog au moyen de la commande *reload* (voir ci-dessus).

4. Entrées / sorties

Tous les organes d'entrée / sortie (clavier, écran, fichiers, etc…) utilisés par un programme Prolog III y sont représentés par des entités appelées *unités d'entrée/sortie*. La représentation écrite d'une telle unité est une chaîne de caractères ; le cas échéant, cette chaîne représente aussi une information identifiant l'unité vis-à-vis du système d'exploitation : nom de fichier, etc…

A tout moment, le système Prolog III «connaît» un certain nombre d'unités d'entrée/sortie : ce sont les unités qui ont été précédemment ouvertes et qui n'ont pas encore été fermées. Les descripteurs de ces unités sont rangés dans une pile : l'unité qui est au sommet s'appelle l'unité courante. Toutes les opérations d'entrée / sortie s'effectuent sur l'unité courante. Au lancement, une unité d'entrée (appelée *console*) et une unité de sortie (appelée également *console*) sont ouvertes automatiquement par le système et sont les unités courantes ; habituellement, ces unités sont associées au clavier et à l'écran de l'ordinateur ou du terminal depuis lequel Prolog a été lancé.

Changer d'unité (c'est-à-dire exécuter les primitives *input(f)* ou *output(f)*) ne ferme pas l'unité courante ni ne change son état, mais ouvre -si nécessaire- une nouvelle unité et la met au sommet de la pile, au-dessus de la précédente unité courante, de façon à pouvoir restaurer cette dernière lors de la fermeture de la nouvelle unité courante.

Entrées

in_char(*t*)
Lecture d'un caractère.

Lit le prochain caractère, le transforme en une chaîne de longueur 1 et essaie de l'unifier avec *t*.

Exemple :

```
> in_char(c1)  in_char(c2)  in_char(c3);
ABC
{c1 = `\12`, c2 =  `A`, c3 = `B` }
>
```

Dans cet exemple, le caractère obtenu lors de la première lecture est en réalité le caractère fin-de-ligne qui terminait la ligne de commande.

in_term(*t*)
Lecture d'une suite de termes Prolog III.

Lit la plus grande suite de caractères qui constituent l'expression écrite (syntaxiquement correcte) d'une suite de termes Prolog III ; ensuite, tente l'unification de *t* avec le premier des termes lus. Les caractères blancs qui suivent le dernier terme lu sont lus eux aussi.
Exemple:

```
> in_term(t)  in_char(c);
nom_marie_poids("Dupont",1',755/10) est un terme;
{ t = nom_marie_poids("Dupont",1',151/2)  c = `\12` }
>
```


© PrologIA

in_sentence(t_1, t_2, t_3)
Lit une phrase qui se termine par ".", "?" ou "!" et la met sous la forme de trois listes, respectivement unifiées avec t_1, t_2 et t_3. t_1 est la liste des unités lexicales (mots, nombres, caractères spéciaux, etc…) constituant la phrase ; t_3 est la liste obtenue en remplaçant chaque lettre majuscule apparaissant dans un élément de t_1 par la lettre minuscule correspondante. t_2 est la même chose que t_3, mais chaque élément de t_3 qui correspond à un identificateur connu dans la famille du préfixe par défaut du contexte courant a été remplacé par le symbole conventionnel <> pour indiquer ce fait.

Exemple :

```
> truc;
> in_sentence(t1,t2,t3);
truc, Truc et line sont connus.
{t1 = <"truc",",","Truc","et","line","sont","connus",".">,
 t2 = <<>,",",<>,"et","line","sont","connus",".">,
 t3 = <"truc",",","truc","et","line","sont","connus",".">}
>
```

input(*u*)
L'unité dont le nom est *u* devient l'unité d'entrée courante ; elle ne doit pas être une des unités précédemment ouvertes et non encore fermées. Le fichier correspondant est recherché et ouvert.

Aucune autre action, en plus du changement de l'entrée courante, n'est exécutée pour le moment ; en particulier, Prolog III ne bascule pas dans un quelconque « mode lecture », mais continue l'exécution du programme en cours. Ce n'est que lorsque ce programme effectuera des lectures sur l'unité courante, ou bien lorsqu'il sera terminé et que Prolog III retournera dans l'état « lecture d'une commande », que l'effet de *input(u)* se manifestera, puisque les lectures seront faites à partir de l'unité indiquée et non sur l'unité courante.

Bien entendu, un fichier ouvert par *input* peut contenir aussi bien des règles que des requêtes. De plus, il peut lui-même contenir une commande *input*, la nouvelle unité courante est empilée au-dessus de la précédente.

close_input

L'unité courante est fermée et son descripteur est enlevé du sommet de la pile des unités : l'unité précédente redevient unité courante.

Sorties

out(*t*)
outl(*t*)
Ecrit le terme *t* sur l'unité courante. La lecture par *in_term* de la forme écrite d'un terme *t* produite par *out* reconstruit le terme *t* original.

Quelque soit le type du terme à écrire, si son expression est plus longue que la place restant sur la ligne courante, alors elle est coupée à l'endroit requis par insertion d'un retour-chariot *masqué* (par un caractère «\»). L'écriture du terme obéit en outre aux conventions suivantes:

- Les listes sont imprimées dans la syntaxe d'Edimbourg (crochets [ ]).
- Tout identificateur ne répondant pas à la syntaxe d'un identificateur est quoté.
- Le contexte courant détermine les abréviations possibles pour les identificateurs.

outl(t) équivaut à la suite de buts *out(t) line* : lorsque le terme *t* a été écrit on produit le passage à la ligne suivante.

Exemple :

```
> out([1, Pierre, "Salut!"]);
[1, Pierre, "Salut!"]{}
>
```

N.B. : Les accolades imprimées à la suite du terme indiquent la réussite de l'effacement du but `out([1, Pierre, "Salut!"])`. Une manière d'en empêcher l'impression consiste à faire échouer artificiellement cet effacement :

© PrologIA

```
> out([1, Pierre, "Salut!"])  line  fail;
[1, Pierre, "Salut!"]
>
```

outm(*s*)
outml(*s*)
Ecrit la chaîne *s* sur l'unité de sortie courante, sans écrire les quotes et en interprétant correctement les caractères de formatage que *s* peut contenir (comme « \n », etc…). Si la chaîne est plus longue que la place restante sur la ligne en cours, alors elle est coupée par un retour chariot «masqué». *outml(s)* équivaut à *outm(s) line*.

Exemple:

```
> out("\tBonjour!")  line  fail;
"\tBonjour!"
> outm("\tBonjour!")  line  fail;
     Bonjour!
>
```

outc(*t*)
Ecrit le terme *t* et les contraintes qu'il supporte.

Exemples :

```
> outc(t) fail , { t ::num };
t,
   {t::num  }
> outc(t) fail , { t ::numt };
X3[X2],
   {X3::num  }
>
```

line
Ecrit une marque de fin-de-ligne sur l'unité de sortie courante.

output(*u*)
L'unité de nom *u* devient l'unité courante de sortie. Le cas échéant, un fichier de texte est créé, ayant le nom indiqué par la chaîne *u*.

close_output
L'unité courante est enlevée du sommet de la pile des unités de sortie ouvertes. Si elle correspond à un fichier, alors celui-ci est fermé.

echo
no_echo
Active (resp. annule) l'option qui provoque l'affichage sur l'unité *console* des termes lus ou écrits sur une autre unité.

ATTENTION : les termes sont d'abord lus et codés, puis décodés et affichés. Ce que vous voyez est donc équivalent à ce qui a été lu, mais peut en différer de manière plus ou moins importante. L'intérêt de cette façon de procéder réside dans le fait que vous voyez les termes non pas comme vous les avez tapés, mais comme Prolog III les a "compris" ce qui, dans une situation de recherche d'erreur, est préférable.

© PrologIA

5. Autres éléments de l'environnement

Sortir de Prolog III

exit
exit(*s*)
Fait sortir de Prolog en sauvant la totalité des programmes couramment connus, y compris ceux qui constituent le superviseur de Prolog III. Le fichier produit a pour nom *prolog3.psv* (première forme) ou la valeur de s, qui doit être une chaîne de caractères (deuxième forme).

quit(*n*)
quit
Fait quitter Prolog III sans rien sauver. Si *n* a pour valeur 0, cette valeur est convertie dans le code correspondant à «pas d'erreur» pour le système d'exploitation; les autres valeurs de *n* ne sont pas converties. Ce code est ensuite envoyé au système d'exploitation à titre de “statut de terminaison”. La deuxième forme, *quit*, équivaut à *quit(0)*.

Temps et autres mesures

reset_cpu_time
réinitialise le chronomètre.
cpu_time(*x*)
cpu_time unifie *x* avec le temps cpu en millisecondes écoulé depuis le dernier *reset_cpu_time*.

ststs

Association Prolog
HERITAGE

Règles prédéfinies et procédures externes

Que trouve-t-on dans ce chapitre ?

Ce chapitre regroupe par ordre alphabétique toutes les règles prédéfinies et procédures externes disponibles en Prolog III.
On retrouvera la plupart des primitives de Prolog II et Prolog II+, ainsi qu'un grand nombre de nouvelles règles prédéfinies adaptées aux fonctionnalités spécifiques de Prolog III. La plupart de ces prédicats ont déjà été abordés dans les chapitres précédents. Un accès thématique à ce dictionnaire des primitives Prolog III est également fourni à la fin du manuel.

© PrologIA

1. Introduction

Ce chapitre présente les règles prédéfinies et procédures externes de Prolog III. Cette partie du manuel est certainement celle qui demeurera, au fur et à mesure de vos développements, la plus utile. C'est pourquoi nous avons essayé de vous faciliter la tâche en donnant, pour chacune de ces primitives un mini mode d'emploi, et en multipliant les accès pertinents au reste du manuel.

Ainsi, pour chacun de ces prédicats, on trouvera les références suivantes :

Fonction : décrit en quelques mots les fonctionnalités du prédicat

Classe : détermine s'il s'agit d'une procédure externe, d'une règle prédéfinie ou d'une fonction évaluable, et donne les compatibilités Prolog II et Edimbourg.

Catégorie : renvoie généralement au chapitre concernant ce type de prédicat, et donne souvent une sous-catégorisation.

Paramètres connus : informe sur la nécessité que certains des arguments soient connus à l'appel de la primitive concernée.

Message d'erreur : renseigne sur la présence éventuelle d'un ou plusieurs messages d'erreurs attachés à une mauvaise utilisation de la primitive. A noter que dans certains cas, on préfère échouer sans imprimer de message d'erreur.

De plus, les arguments pour chacune de ces primitives correspondent, dans la plupart des cas, au type requis (**N** pour entier, **S** pour string, **T** pour terme, **R** pour rationnel, **V** pour variable, **P** pour prédicat, etc ...).

© PrologIA

On donne ensuite une description plus détaillée de la primitive, suivie d'exemples de sessions qui la font directement intervenir, éventuellement un certain nombre de remarques, et pour finir une liste de primitives à consulter dont les fonctions se rapprochent de celle considérée.

Ce chapitre est séparé en deux parties distinctes. Une partie décrivant les primitives utilisables en syntaxe standard, et une partie décrivant les “primitives Edimbourg”.

On trouvera également à la fin de chacune de ces sous-parties un index spécifique concernant les règles prédéfinies, les procédures externes, les fonctions évaluables, ainsi que certains identificateurs importants rencontrés tout au long de ce chapitre.

add_implicit(S1,S2)

Fonction	: Ajout d'un identificateur
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *S1* et *S2*
Message d'erreur	: Si *S1* n'est pas le préfixe d'une famille ayant une partie fermée

Description :

Rajoute l'identificateur de nom abrégé *S2* à la partie fermée du contexte *S1*. Si *S1* n'est pas le préfixe d'une famille ayant une partie fermée, on obtient une erreur.

Voir la section « *Structuration, saisie et modification des règles* » du chapitre « *Le contrôle et l'environnement des programmes* » pour plus de détails.

Voir également :

- *remove_implicit, dictionary*
- *set_context, current_context, close_context_dictionary*
- *module, end_module*

© PrologIA

arg(N, T1, T2)

Fonction	: Calcul du N-ième argument d'un terme
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: *N*, qui doit être un entier positif ou nul
Messages d'erreur	: *N* est inconnu, négatif ou plus grand que le nombre d'arguments de *T1*

Description :

Cette primitive pose la contrainte *{T2 = N'}*, où *N'* est le *N-ième* argument du terme *T1*. Si *N* est nul, *T2* est égal à l'étiquette initiale de *T1*.

Exemples :

```
> arg(2,exemple(X,Y,5),T);
  { T = Y }
> arg(3,exemple[L],T);
  { L = V.<T>.V',
    V :: 2 }
> arg(0,exemple[L],T);
  { T = exemple, L !tuple }
> arg(5,exemple(X,Y,5),T);
  ERREUR 228: Argument de mauvais type
>
```

```
> arg(3,U,T), {U :: tuple};
  { U = V.<T>.V',
    V :: 2 }
> arg(2,[A,B,C],T);
  { T = [B,C] }
>
```

Voir également :

- *arg2, arg3*

arg2(N, L, T)

Fonction	: Calcul du N-ième élément d'une liste
Classe	: Compatibilité partielle Prolog II+
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: *N*, qui doit être un entier positif ou nul
	L qui doit être une liste suffisamment connue
Messages d'erreur	: *N* est inconnu, négatif ou plus grand que le nombre d'éléments de *T*.
	La liste *L* n'est pas suffisamment connue

Description :

Si *N* est nul, *arg2* pose la contrainte *{T = S}*, où *S* représente le nombre d'éléments de la liste *L*

Si N n'est pas nul, *arg2* pose la contrainte *{T = X}*, où *X* est le *N-ième* élément de la liste *L*.

Exemples :

```
> arg2(4,[1,2,3,4,5],T);
  { T = 4 }
> arg2(2,[A,B,C|L],T);
  { B = T }
> arg2(3,[A|L],T);
  ERREUR 228: Argument de mauvais type
>
```

Voir également :

- *arg, arg3*

© PrologIA

arg3(N, T1, T2)

Fonction	: Calcul du N-ième argument d'un tuple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: *N*, qui doit être un entier positif ou nul
Messages d'erreur	: *N* est inconnu, négatif ou plus grand que le nombre d'arguments de *T1*

Description :

Cette primitive pose la contrainte *{T2 = N'}*, où *N'* est le *N-ième* argument du tuple *T1*. Si *N* est nul, *T2* est égal à l'étiquette initiale de *T1* c'est-à-dire **<>**. Echoue si *N* n'est pas un entier positif connu ou si *T1* n'est pas un tuple.

Exemples :

```
> arg3(2,<X,Y,5>,T);
  { T = Y }
> arg3(3,L,T);
  { L = <T>.V }
> arg3(0,E[L],T);
  { E = <>, T = <>, L !tuple }
> arg3(5,<X,Y,5>,T);
  ERREUR 228: Argument de mauvais type
> arg3(5,exemple(1,2,3),T);
>
```

Voir également :

- *arg, arg2*

assert(T, Q)

Fonction	: Ajout d'une règle au début de son paquet
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T* et *Q*
Messages d'erreur	: - si *Q* n'est pas une liste "classique" - si *T* est une tête de règle incorrecte

Description :

Ajout d'une règle, au début de son paquet. *T* doit être un terme pouvant être une tête de règle, c'est-à-dire représentant un arbre étiqueté par un identificateur et *Q* doit être une liste de termes.

L'effacement de *assert(T, Q)* ajoute la règle *T -> Q au-dessus* du groupe de règles ayant le même «nom» que *T*, c'est-à-dire au début du paquet correspondant à *T*. Par exemple, les deux commandes suivantes

Exemples :

```
> assert(conc([e|x],y,[e|z]),[conc(x,y,z)]);
  {}
> assert(conc([],y,y),[]);
  {}
>
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
conc([],y,y) ->;
conc(conc([e|x],y,[e|z]) -> conc(x,y,z);
```

Il semble à première vue que *assert* ne permet pas d'ajouter des règles comportant un système de contraintes. Il n'en est rien, comme le montre l'exemple suivant :

© PrologIA

Exemples :

```
> assert(hors_de(x,[y|q]),[hors_de(x,q)],  {x # y};
  { x # y }
> assert(hors_de(x,[]),[]);
  {}
> list;
hors_de(x,[]) -> ;
hors_de(x,[y | q]) ->
   hors_de(x,q),
   { x # y } ;
  {}
>
```

asserta(T,Q)

Fonction	:	identique à assert (compatibilité Edinburgh)
Classe	:	Règle prédéfinie Prolog III (Edinburgh)
Catégorie	:	Gestion des règles et des identificateurs
Paramètres connus	:	*T* et *Q*
Messages d'erreur	:	- si *Q* n'est pas une liste “classique” - si *T* est une tête de règle incorrecte

Identique à *assert(T, Q)*

assert''(T, Q)

Fonction	: Ajout d'une règle la fin de son paquet
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T* et *Q*
Messages d'erreur	: - si *Q* n'est pas une liste “classique” - si *T* est une tête de règle incorrecte

Description :

Ajout d'une règle, à la fin de son paquet. Même fonctionnement que *assert*, mais l'ajout se fait au-dessous, et non au-dessus, du paquet correspondant à *T*. Exemple: les deux commandes suivantes, dans l'ordre indiqué, produisent l'insertion du même programme *conc* que ci-dessus :

Exemples :

```
> assert''(conc([],Y,Y),[]);
  {}
> assert''(conc([E|X],Y,[E|Z],[conc(X,Y,Z)]);
  {}
>
```

assertz(T, Q)

Fonction	: identique à assert'' (compatibilité Edinburgh)
Classe	: Règle prédéfinie Prolog III (Edinburgh)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T* et *Q*
Messages d'erreur	: - si *Q* n'est pas une liste “classique” - si *T* est une tête de règle incorrecte

Identique à *assert''(T, Q)*

© PrologIA

assign(I, T)

Fonction	: Affectation
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *T* et *I*
Messages d'erreur	: Si *T* ne représente pas un identificateur, un nombre, un caractère ou une chaîne de caractères.

Description :

Affecte le terme *T*, qui doit représenter un identificateur, un nombre, un caractère ou une chaîne de caractères, à l'identificateur *I*. Tout se passe comme si *I* devenait le nom d'une variable «globale» (ultérieurement accessible pendant l'effacement de n'importe quel but) et «statique» (résistante au *backtracking*) possédant *T* pour valeur. Il s'agit donc bien de l'affectation classique, comme elle se pratique en FORTRAN, Pascal, etc….

Exemple :

```
> assign(nom_fichier,   "monfichier.txt");
  {}
> val(nom_fichier,x);
  { x="monfichier.txt" }
>
```

En Prolog, ces variables statiques peuvent être vues comme une manière particulièrement efficace d'écrire des *assertions* (ou règles sans queue). Du point de vue de l'utilisateur, on peut considérer que l'emploi de *assign* et *val* fait ci-dessus est équivalent à :

```
> retract(nom_fichier(x),  [] );
  {}
> assert(nom_fichier("monfichier.txt"),  [] );
  {}
> nom_fichier(x);
  { x="monfichier.txt" }
>
```

assign(tab(N), T)

Fonction	: affectation d'un élément d'un tableau
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *tab(N)* et *T*
Messages d'erreur	: Entier attendu, Tableau déjà défini, Tableau non défini, Débordement de tableau.

Description :

Affecte la valeur de *T* à l'élément de rang *N* du tableau *tab*. Ce tableau doit avoir été défini au préalable avec *def_array*. Les mêmes restrictions que précédemment s'appliquent à *T* : ce doit être une constante.

Voir également :

- *val*
- *def_array, undef_array, redef_array*

© PrologIA

block(E, C, B), block_exit(E, C)

Fonction	: Abandon d'effacement, Récupération d'erreurs
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *B*
Messages d'erreur	: Block indéfini

Description :

La règle prédéfinie *block* permet de terminer brutalement l'effacement d'un but, ou d'une liste de buts, *B*. Cette primitive est essentiellement utilisée pour la récupération des erreurs ou le contrôle de programme.

On peut considérer que :

- Pour effacer *block(E, C, B)* on efface *B* en ayant auparavant créé une paire de parenthèses fictives, étiquetées par *E* et éventuellement par *C* , autour du but *B*.

- Le deuxième paramètre *C*, de *block(E C, B)* , peut être considéré soit comme une seconde étiquette, soit comme un complément d'information qui sera retourné par *block_exit(E, C)*. Certaines règles prédéfinies, utilisent ce deuxième paramètre comme complément d'information, en transmettant l'argument cause de l'erreur.

- *block_exit(E, C)* provoque l'abandon immédiat de l'effacement de tous les buts inclus entre les parenthèses étiquetées par *E* et *C*. L'effacement continue ensuite, comme si *block(E, C, B)* s'était effacé, mais sans laisser aucun choix pour le ou les buts *B* en attente.

De plus :

- Les étiquettes dont il est question ci-dessus sont des termes Prolog III quelconques, et on considère que pour que deux termes représentent la même étiquette au regard du mécanisme *block_exit*, il faut et il suffit qu'ils soient unifiables.

- S'il y a plusieurs parenthèses étiquetées par *E* et *C*, *block_exit(E, C)* s'arrête au couple de parenthèses le plus interne.

- Si *block_exit(E, C)* ne rencontre pas de parenthèses *E* et *C* on revient au niveau de commande avec, si *E* n'est pas un entier, le message d'erreur “Erreur 50 : Block indéfini”, ou le message correspondant à l'erreur *E* si *E* est un entier.

C'est ce même mécanisme qui est utilisé pour la gestion interne des erreurs en Prolog III. Une erreur rencontrée dans l'exécution d'un programme provoque l'effacement du but *block_exit(I, C)* où *I* est un entier correspondant au numéro de l'erreur et *C* un complément d'information ou le tuple vide. De cette manière, l'utilisateur a la possibilité de récupérer toutes les erreurs de Prolog III pour les traiter dans son application.

Exemples :

```
executer_commande ->
  block(fin_commande, C, lire_et_executer);

lire_et_executer ->
  repeat
  outm("? ")
  in_char'(X)
  executer(X);

executer(`t`) ->                  % terminaison de programme
  block_exit(fin_commande, <>);
executer(`c`) ->
  outml("continuer");
executer(X) ->
  block_exit(fin_commande, X),
  { X # `c` };
```

Dans cet exemple, la commande `t` utilise *block_exit* pour revenir au niveau supérieur de *executer_commande* en interrompant l'effacement de *lire_et_executer* et en supprimant tous les points de choix (*repeat*) en attente. La commande `c` termine normalement l'effacement de *lire_et_executer* et conserve les points de choix (*repeat*). Toute autre commande utilise aussi *block_exit* et permet de remonter le caractère tapé comme complément d'information pour *C*.

© PrologIA

```
lecture_dans_un_fichier(L) ->
   input("fichier_a_lire")
   boucle_de_lecture(L);

boucle_de_lecture(L) ->
   block(X, _, inl(D_onnee))
   suite_boucle_de_lecture(X, D_onnee, L);

suite_boucle_de_lecture(X, D_onnee, <D_onnee>.L) ->
   free(X)  !
   boucle_de_lecture(L);
suite_boucle_de_lecture(17, _, <>) ->  % fin de fichier
   close_input("fichier_a_lire");
```

Ce deuxième exemple permet de lire des données (chaînes de caractères) dans un fichier et de construire la liste (tuple) de ces données. Lorsque l'interprèteur Prolog détecte la fin du fichier, il génère un *block_exit*, avec l'entier 17 comme valeur de l'étiquette principale, correspondant à l'erreur de fin de fichier. Cette erreur peut être récupérée par le *block* englobant puisque son étiquette principale était une variable.

```
lecture_dans_un_fichier_et_traitement ->
   input("fichier_a_lire")
   block(E, C, lire_et_traiter(I_nfo))
   suite_lire_et_traiter(E, C, I_nfo);

lire_et_traiter(I_nfo) ->
   repeat
   inl(I_nfo)
   traiter(I_nfo);

suite_lire_et_traiter(X, _, _) ->
   free(X)  !
   fail;
suite_lire_et_traiter(17, _, _) -> !   % fin de fichier
   close_input("fichier_a_lire");
suite_lire_et_traiter(X, Y, _) -> !    % erreurs non récupérées
   close_input("fichier_a_lire")
   block_exit(X, Y),
   { X <= 9 };
suite_lire_et_traiter(X, Y, I) ->      % erreurs traitées
   traiter_erreur(I, X, Y)
   block(E, C, lire_et_traiter(I_nfo))
   suite_lire_et_traiter(E, C, I_nfo);

traiter_erreur(I, X, Y) ->
   ... ;

traiter(I) ->
   ... ;
```

Le dernier exemple permet de lire des données (chaînes de caractères) dans un fichier et de les traiter sans construire la liste (tuple) de ces données. Le même mécanisme que précédemment est utilisé lorsque Prolog détecte la fin de fichier, et un mécanisme analogue permet de gérer les autres erreurs pouvant intervenir au cours du traitement. Une partie de ces erreurs ne sont pas récupérables, telles que les erreurs de débordement de piles, et dans ce cas on utilise *block_exit* pour retourner au niveau supérieur de Prolog. Par contre, certaines autres erreurs peuvent être prises en compte dans le traitement afin de poursuivre l'exécution du programme, il faut dans ce cas relancer l'effacement d'un nouveau *block* puisqu'il n'y a plus aucun moyen de revenir dans le *block* précédant.

Remarque :

A tout instant un programme peut être interrompu au moyen d'une touche déterminée, dépendant du système utilisé, par exemple : <Ctrl-C>. Cette interruption est gérée soit comme une erreur, il est alors possible de l'intercepter par le mécanisme de *block*, soit comme un dialogue, et dans ce cas *block* ne peut l'intercepter. L'utilisateur peut configurer la gestion de l'interruption à l'aide de la procédure externe *set_config*.

Voir également :

- *set_config*

© PrologIA

block(E, B), block_exit(E)

Fonction	: Abandon d'effacement, Récupération d'erreurs
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *B*
Messages d'erreur	: Block indéfini

Description :

Fonctionnement analogue aux formes précédentes, avec une «étiquette» *E* à la place de deux. *B* est le but ou la suite de buts à effacer. *block(E, B)* lance l'effacement de *B* ; l'effacement ultérieur de *block_exit* à un argument produira la recherche d'un *block* dont le premier argument s'unifie avec l'argument de *block_exit*.

- La forme *block(E, B)* est équivalente à *block(E, _, B)*.
- La forme *block_exit(E)* est équivalente à *block_exit(E, <>)*.

Lorsque *block_exit(E, C)* est effacé dans un environnement «parenthésé» par *block(E, B)*, alors *block_exit(E, C)* se comporte comme *block_exit(E)*. Inversement, lorsque *block_exit(E)* est effacé dans un environnement «parenthésé» par *block(E, C, B)*, alors *block_exit(E)* se comporte comme *block_exit(E, <>)*.

Remarque :

A tout instant un programme peut être interrompu au moyen d'une touche déterminée, dépendant du système utilisé, par exemple : <Ctrl-C>. Cette interruption est gérée soit comme une erreur, il est alors possible de l'intercepter par le mécanisme de *block*, soit comme un dialogue, et dans ce cas *block* ne peut l'intercepter. L'utilisateur peut configurer la gestion de l'interruption à l'aide de la procédure externe *set_config*.

Voir également :

- *set_config*

bool(B)

Fonction	: Vérification booléenne
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: *B*
Messages d'erreur	: Aucun

Description :

Cette règle prédéfinie vérifie que *B* est un terme représentant une valeur booléenne connue. Si ce n'est pas le cas *bool(B)* échoue.

Exemples :

```
> bool(1');
  {}
> bool(A&B);
>
> bool(A|1');
  { A !bool }
> bool(A),{A|B = 1', A|~B = 1'};
  { A = 1',
    B !bool }
> bool(2);
>
```

Remarques :

Cette règle prédéfinie ne teste pas seulement que *B* représente une valeur booléenne (la règle prédéfinie *is_bool(B)* effectue cette vérification).

Le terme *B* peut être contraint à représenter une valeur booléenne connue dans le cas où, pour toutes les solutions du système de contraintes courant, *B* représente la même valeur (voir le troisième exemple ci-dessus)

© PrologIA

bound(T)

Fonction	: Vérifie que *T* représente un terme suffisamment "connu"
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ce prédicat s'efface si *T* est un terme représentant un arbre dont :

- L'étiquette initiale est connue
- Le nombre des fils est connu

Voir la section « *Termes connus* » du chapitre « *Retardements* » pour plus de détails.

Exemples :

```
> bound(1');
  {}
> bound(123/456);
  {}
> bound(aurore(A,B,C,D));
  {}
> bound(boreale[L]),{L  ::  12};
  { L :: 12, L !tuple }
> bound(E[L]),{L  ::  12};
>
```

Voir également :

- *known, bound_tree, bound_tree'*
- *free, free_label'*

bound_conc(U1, U2, U3)

Fonction	: concaténation
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Pose la contrainte *{U3 = U1.U2}*. Echoue si la taille du tuple *U1* n'est pas connue.

Exemples :

```
> bound_conc(<1,2>.X,Y,Z),  {  X::3  };
  { Z = <1,2>.X.Y, X::3 }
> bound_conc(X,Y,Z);
>
```

Voir également :

- *conc3*

© PrologIA

bound_mult(N1, N2, N3)

Fonction	: Multiplication linéaire
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N1*N2}*. Echoue si *N1* et *N2* ne sont pas des valeurs numériques connues (en d'autres termes réussit si l'une des deux est connue, ce qui rend la contrainte linéaire).

Exemples :

```
> bound_mult(2,Y,Z);
  { Z = 2Y }
> bound_mult(X,X,4);
>
```

Voir également :

- *mult*

bound_size(T, N)

Fonction	: Taille d'un terme
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: N ou la taille de T (voir ci-dessous)
Messages d'erreur	: Non

Description :

Pose la contrainte *{ T :: N }*, si *N* est un entier positif, ou bien la contrainte *{ N = k }* si *T* est un terme ayant un nombre connu, *k*, de fils. Echoue dans les autres cas.

Exemples :

```
> bound_size(<1,2,3,4>,N);
  { N = 4 }
> bound_size("c'est  tout!",N);
  { N = 11 }
> bound_size(U,2);
  { U = X1[X2],
   X2 :: 2 }
> bound_size(U,N);
>
```

Dans cet exemple nous avons renommé les variables intermédiaires créées par Prolog III.

© PrologIA

bound_tree(T)

Fonction	: Vérifie que *T* représente un arbre unique
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que *T* représente un arbre unique, c'est à dire que le terme *T* ne comporte aucune variable.

Remarque :

T ne doit pas représenter un arbre infini. Si c'est le cas, utiliser la primitive *bound_tree'*.

Exemples :

```
> bound_tree(<1,2,3,4>);
  {}
> bound_tree("c'est  tout!");
  {}
> bound_tree(E[U])  {  U::4  };
>
```

Voir également :

- *free, free_label, known, bound, bound_tree'*

bound_tree'(T)

Fonction	: Vérifie que *T* représente un arbre unique, éventuellement infini
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que *T* représente un arbre unique, c'est à dire que le terme *T* ne comporte aucune variable. *T* peut représenter un arbre infini. Si on est sûr d'être dans le cas contraire, utiliser la primitive *bound_tree*, plus efficace.

Exemples :

```
> bound_tree'(<1,2,3,4>);
  {}
> bound_tree'(X),{X  =  1(X)};
  { X = 1(X) }
>
```

Voir également :

- *free, free_label, known, bound, bound_tree*

© PrologIA

break
break(S1)
break(S1, S2)

Fonction	: Appel au debugger Prolog III
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *S1* et *S2*
Messages d'erreur	: non

Description :

Cette primitive a pour effet de passer en mode `debug` et, selon la présence d'arguments, d'imprimer la chaîne *S1* sur la sortie de trace, puis d'exécuter sous le debugger Prolog III les commandes contenues dans chaîne *S2*.

Placer des appels à *break* dans un programme revient à installer manuellement des points-espions à des endroits beaucoup plus précis que ceux placés au moyen de la primitive Prolog *spy* sur tout un paquet de règles. De plus, la possibilité de passer des commandes au debugger par l'argument *S2* permet une certaine automatisation (*batch*).

Si on n'est pas en mode `debug` au moment de l'appel à *break*, la ou les commandes *S2* ne seront prises en compte et exécutées qu'à partir d'un prochain port (éventuellement pendant un backtracking). Sinon le port `EXIT` de *break* est montré et les éventuelles commandes *S2* sont exécutées.

Voici un exemple tiré de *menu.p3* , dans lequel on a remplacé le paquet de nom *Poisson* par ce qui suit :

```
Poisson(sole, 2) -> break("dans Poisson", "sol;p;C");
Poisson(thon,4) ->;
```

On souhaite donc espionner la première règle du paquet *Poisson*, et à chaque invocation de cette règle, imprimer un message (*"dans Poisson"*), afficher le système de contraintes portant sur les variables de la question (**sol**), imprimer le status du port courant (**p**) et enfin poursuivre l'exécution jusqu'au prochain point d'arrêt (**C**).

Voici un exemple d'exécution dans lequel ce qui est en gras est tapé par l'utilisateur. Les impressions provoquées par l'appel à *break* sont en italique. On suppose que l'on n'est pas en mode `debug` au moment de lancer la requête :

```
> RepasLeger(h,p,d);
{h = radis, p = boeuf, d = fruit}
{h = radis, p = porc, d = fruit}
dans Poisson
  6[0]CALL    : Dessert(d,k_1)
{h = radis, p = sole}
  6[0]CALL    : Dessert(d,k_1)
{h = radis, p = sole, d = fruit}
{h = radis, p = sole, d = glace}
{h = radis, p = thon, d = fruit}
dans Poisson
    5[2]EXIT(rC): break("dans Poisson","sol;p;C")
{h = pate, p = sole}
    5[2]EXIT(rC): break("dans Poisson","sol;p;C")
{h = pate, p = sole, d = fruit}
>
```

L'utilisateur se réfèrera au manuel “Mise au point de programmes Prolog III” pour la définition des points-espion.

Voir également :

- *spy*
- *debug, no_debug*

© PrologIA

char(C)

Fonction	: Vérifie que *C* représente un caractère connu
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: *C*
Messages d'erreur	: Non

Définition :

Le but *char(C)* s'efface si C est un terme représentant un caractère connu.

Exemples :

```
> char(`a`);
  {}
> char(A),  {A  !char};
>
  char("A");
>
```

char_code(C, N)

Fonction	: Liaison d'un caractère et de son code ASCII
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversion
Paramètres connus	: *C* ou *N*
Messages d'erreur	: Non

Définition :

Fait correspondre au caractère *C* son code ASCII *N* et vice-versa.

Exemple :

```
> char_code(`A`,N);
  {N=65}
> char_code(C,65);
  {C=`A`}
>
```

Remarque :
Les caractères dont le code ASCII est compris entre 0 et 31 sont imprimés sous la forme `` `\uu` `` où `uu` représente l'écriture octale du code en question.

© PrologIA

close_context_dictionary(S)

Fonction	: Fermeture d'une famille d'identificateurs
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *S*
Messages d'erreur	: Non

Description :

Définit la *partie fermée* de la famille *S*, comme étant égale à l'état courant de cette famille au moment où cette règle est exécutée. Tout autre identificateur venant s'ajouter à cette famille par la suite ne sera pas dans sa partie fermée et donc ne sera pas pris en compte dans les opérations utilisant la « liste implicite » de la commande *set_context* (voir cette primitive), sauf s'il fait l'objet d'un appel de la règle *add_implicit*..

Voir également :

- *add_implicit, remove_implicit, dictionary*
- *set_context, current_context*

close_input

Fonction	: Fermeture de l'unité d'entrée courante
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Impossible de fermer le fichier

Description :

L'unité courante est fermée (sauf s'il s'agit de la console) et son descripteur est enlevé du sommet de la pile des unités : l'unité précédente redevient unité courante.

close_input(F)

Fonction	: Fermeture de l'unité d'entrée de nom *F*
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/Sorties
Paramètres connus	: *F*, qui doit être une chaîne de caractères
Messages d'erreur	: Impossible de fermer le fichier

Description :

L'unité de nom *F*, qui doit figurer dans la liste des unités ouvertes, est fermée et son descripteur est enlevé de cette liste.

Voir également :

- *input*

© PrologIA

close_output

Fonction	: Fermeture de l'unité de sortie courante
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Impossible de fermer le fichier

Définition

L'unité courante est enlevée du sommet de la pile des unités de sortie ouvertes, sauf s'il s'agit de la console. Si elle correspond à un fichier, alors celui-ci est fermé.

close_output(F)

Fonction	: Fermeture de l'unité de sortie de nom *F*
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/Sorties
Paramètres connus	: *F*, qui doit être une chaîne de caractères
Messages d'erreur	: Impossible de fermer le fichier

Description :

L'unité de nom *F*, qui doit figurer dans la liste des unités de sortie ouvertes, est fermée et son descripteur est enlevé de cette liste.

Voir également :

- *output*

conc3(U1, U2, U3)

Fonction	:	Concaténation retardée de tuples avec contraintes arithmétiques sur les tailles.
Classe	:	Règle prédéfinie Prolog III
Catégorie	:	Arbres, listes chaînes et tuples
Paramètres connus	:	Voir ci-après
Messages d'erreur	:	Non

Description :

Pose le système de contraintes *{U3 = U1.U2, U1 :: N1, U2 :: N2, U3 :: N3, N3 = N1+N2, N1 >= 0, N2 >= 0, N3 >= 0 }*, avec les retardements nécessaires si *N1, N2 ou N3* ne sont pas connus.

Exemples :

```
> conc3(U,V,U);
  { V = <>,
   U :: N,
   U.<> = U,
   N >= 0 }
>
```

Voir également :

- *size, bound_size, bound_conc*

© PrologIA

conc_string(S1, S2, S3)

Fonction	: Concaténation de chaînes
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Arbres, listes chaînes et tuples
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Enumère tous les triplets *S1*, *S2*, *S3* de chaînes telles que *S3* est la concaténation de *S1* et *S2*. Les chaînes *S1*, *S2*, *S3* doivent être suffisamment connues pour produire des ensembles finis de triplets.

Exemples :

```
> conc_string("ab","cd",s);
  { s="abcd" }
> conc_string(s1,s2,"abcd");
  { s1=<>,s2="abcd" }
  { s1="a",s2="bcd" }
  { s1="ab",s2="cd" }
  { s1="abc",s2="d" }
  { s1="abcd",s2=<> }
>
```

cpu_time(N)

Fonction	: Obtention d'un « temps cpu »
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

cpu_time unifie *N* avec le temps cpu en millisecondes.

Voir également :

- *reset_cpu_time*

© PrologIA

current_context(T0), current_context(T0, T1, T2, T3)

Fonction	: Identification de contexte
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Unifie *T0* avec le terme identifiant le contexte courant de lecture écriture, et *T1*, *T2*, *T3* avec, respectivement, le terme définissant la suite explicite, le terme définissant la suite implicite, et le préfixe par défaut de ce contexte.

Voir la section « *Structuration, saisie et modification des règles* » du chapitre « *Le contrôle et l'environnement des programmes* » pour plus de détails.

Voir également :

- *set_context*

current_predicate(<I, A>)

Fonction	: Test de présence d'une règle
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Permet de tester la présence d'une règle. S'efface s'il existe une règle d'identificateur d'accès *I* et d'arité *A*. Si *A* est une variable, et *I* est connu, énumère successivement toutes les valeurs de *A* correspondant à une règle d'accès *I*. Si *I* n'est pas connu (i.e. est une variable), unifie l'argument successivement avec toutes les formes *I, A* des règles du module déterminé par le préfixe par défaut du contexte courant.

Voir également :

- *rule*

© PrologIA

/ ! '!'

Fonction	: Coupure
Classe	: Procédure externe Prolog III
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Le prédicat *'!'* s'exécute de la même manière que / (slash) en syntaxe marseillaise ou *!* (cut) en syntaxe Edinburgh, à savoir qu'il coupe tous les choix possibles le précédant dans la règle où il est présent. Ces choix sont : les autres manières d'exécuter la tête de cette règle, et les autres manières d'exécuter les buts de cette règle précédant le *'!'*.

La différence entre ces trois notations est que la forme 'identificateur quoté' (qui est donc un terme) est la seule utilisable à l'intérieur d'un terme (comme argument par exemple) alors que les formes / et *!* ne sont que des parasites employés uniquement dans une suite de buts (queue de règle ou requête).

```
> couleur(rouge)  ->;
> couleur(bleu)  ->;
> taille(grand)  ->;
> taille(petit)  ->;
> choix1(<X,Y>)  ->  couleur(X)  taille(Y);
> choix1("c'est  tout")  ->;
> choix2(<X,Y>)  ->  !  couleur(X)  taille(Y);
> choix2("c'est  tout")  ->;
> choix3(<X,Y>)  ->  couleur(X)  /  taille(Y);
> choix3("c'est  tout")  ->;
> choix4(<X,Y>)  ->  couleur(X)  taille(Y)  '!';
> choix4("c'est  tout")  ->;
```

```
> choix1(U);
  { U = <rouge,grand> }
  { U = <rouge,petit> }
  { U = <bleu,grand> }
  { U = <bleu,petit> }
  { U = "c'est tout" }
> choix2(U);
  { U = <rouge,grand> }
  { U = <rouge,petit> }
  { U = <bleu,grand> }
  { U = <bleu,petit> }
> choix3(U);
  { U = <rouge,grand> }
  { U = <rouge,petit> }
> choix4(U);
  { U = <rouge,grand> }
> choix1(U)  cut;
  { U = <rouge,grand> }
>
```


© PrologIA

debug

Fonction	: Mode debug pour l'exécution d'un programme
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Passage en mode `debug` pour le suivi de l'exécution d'un programme.

L'utilisateur se réfèrera au manuel “Mise au point de programmes Prolog III” pour de plus amples informations.

Voir également :

- *no_debug, trace, no_trace*
- *set_config, set_configuration, get_config, get_configuration,*

def_array(I, N)

Fonction	: Déclaration de tableau statique
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I* et *N*
Messages d'erreur	: Identificateur attendu, Entier attendu, Tableau déjà défini

Description :

Définit dynamiquement un tableau de constantes Prolog de nom *I* et de taille *N*. Ce tableau se comportera comme une variable «globale» (ultérieurement accessible pendant l'effacement de n'importe quel but) et «statique» (résistante au *backtracking*). Les valeurs légales de l'indice sont incluses dans *1..N*.

Si un tableau de même nom existe déjà :

- s'il s'agit d'un tableau de même taille, il ne se passe rien
- si les tailles diffèrent, il se produit une erreur

L'accès et l'affectation sont analogues à ceux des tableaux des autres langages de programmation. Le tableau est désalloué lorsqu'on tue le module auquel il appartient (c.à.d. les règles ayant le même préfixe).

© PrologIA

Exemples

```
> list;
  inc(i) -> val(i,x) assign(i,x+1);

  dec(i) -> val(i,x) assign(i,x-1);

  initialise -> assign(pointeur,1) def_array(pile,100);

  empile(v) ->
    val(pointeur,p)
    !
    assign(pile(p), v)
    inc(pointeur)
    ,{ p <= 100 };
  empile(v) -> outml("débordement de la pile") fail;

  depile(v) ->
    dec(pointeur)
    val(pointeur,p)
    !
    val(pile(p),v)
    dec(pointeur)
    , { p >= 1 };
  depile(v) -> outml("pile vide") assign(pointeur,1) fail;
  {}

> initialise;
  {}
> empile(111);
  {}
> empile(222);
  {}
> depile(x)  depile(y);
  { x=222, y=111 }
> depile(x);
  pile vide
>
```

Voir également :

- *assign, val*
- *undef_array, redef_array*

default(T1, T2)

Fonction	: Contrôle de l'effacement
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Causes d'échec	: *T1* ou *T2* ne représentent pas des arbres étiquetés par un identificateur.

Description :

La règle prédéfinie *default* permet de réaliser le contrôle suivant : si on peut effacer *T1*, alors on l'efface de toutes les manières possibles, sinon on efface *T2*. Il faut remarquer que contrairement à ce que l'on pourrait penser à première vue, cette primitive ne peut pas être réalisée avec '!'.

Exemple :

```
repondre(p) -> default(p,outml("personne"));

homme(jean) ->;
homme(pierre) ->;
```

```
> repondre(homme(x));
  { x=jean }
  { x=pierre }
> repondre(femme(x));
  personne
  {}
>
```


© PrologIA

dictionary

Fonction	: Recherche des modules utilisateurs
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit sur la sortie courante la forme abrégée des accès de la famille ayant le même préfixe que le préfixe par défaut du contexte de lecture/écriture. Chaque accès est écrit sous la forme « *identificateur/arité* »

dictionary(L)

Fonction	: Recherche des modules utilisateurs
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Unifie *L* avec la liste des préfixes des modules utilisateur présents en mémoire.

dictionary(M, L)

Fonction	: Recherche des modules utilisateurs
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *M*, qui doit être une chaîne
Messages d'erreur	: Si *M* n'est pas une chaîne de caractères

Description :

Unifie *L* avec la liste des accès du module *M*. Chaque accès est représenté sous la forme d'un tuple : < *identificateur , arité* >.

Exemple :

```
> aa:bb(1)  ->;
  aa:cc(1,2)  ->;;
> dictionary("aa",L);
  { L = [aa:<bb,1>,aa:<cc,2>] }
>
```


© PrologIA

dif(T1, T2)

Fonction	: Inégalité
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{T1 #T2].*

div(N1, N2, N3)

Fonction	: Division entière
Classe	: Procédure externe Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1* et *N2, entiers. N2 ne doit pas être nul.*
Messages d'erreur	: Division par zéro

Description :

Pose la contrainte *{N3 = D},* où *D* est le résultat de la division entière de la valeur absolue de *N1* par la valeur de *N2.* On prendra garde à ne pas confondre le prédicat avec la fonction évaluable du même nom (voir la règle prédéfinie *val).*

Exemples :

```
>   div(9,2,N);
    { N = 4 }
>   div(7,N1,N2):
>
```

Voir également :

- *mod*
- *trunc*

© PrologIA

dot(T)

Fonction	: Vérifie que T représente une liste binaire non vide
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Vérification de types
Paramètres connus	: *T*, qui doit représenter une liste binaire
Messages d'erreur	: Non

Description :

Vérifie que le terme T représente une paire pointée. Echoue si

T est libre,

ou

T ne représente pas une liste,

ou

T = [].

echo

Fonction	: Echo sur la console
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Provoque l'affichage sur l'unité courante de sortie de toutes les règles ou requêtes lues sur l'unité courante d'entrée. Ces règles lorsqu'elles sont valides sont d'abord codées, puis ensuite listées, donc décodées, sur la sortie courante.

Voir également :

- *no_echo*
- *set_config, set_configuration, get_config, get_configuration*

© PrologIA

edinburgh

Fonction	: Bascule en syntaxe Edimbourg
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Bascule en syntaxe Edimbourg si l'on se trouve en syntaxe Prolog III. Se comporte en fait exactement de la même manière que l'appel :

```
set_config("syntax","Edinburgh")
```

Remarques :

Il est possible de se retrouver directement sous le mode syntaxique Edimbourg, pour chaque session Prolog III. Pour cela, il suffit une première fois d'exécuter la primitive *Edinburgh* (mode syntaxique par défaut : Prolog III), puis de sauver l'état courant à l'aide de la commande *exit*. On invitera chaque utilisateur à renommer auparavant le fichier **initial3.psv** afin d'en conserver une version à l'état d'origine.

Voir également :

- *prologIII*
- *exit, set_config*

edit(I)

Fonction	: Edition de règles
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

La primitive *edit* permet d'éditer un ou plusieurs paquets de règles et réinsère les règles modifiées dans Prolog III, après avoir détruit les règles devenues obsolètes.

L'argument *I* peut avoir toutes les formes décrites pour la primitive *list* c'est-à-dire (*s* désigne une chaîne de caractères, *i* un identificateur, *a* un entier et *v* une variable) :

s	Tous les paquets de règles du module correspondant au préfixe *s*
i	Le(s) paquet(s) de règles ayant *i* pour accès.
$[i_1, i_2, \ldots i_n]$	Tous le(s) paquet(s) de règles ayant i_1 pour accès puis le(s) paquet(s) ayant i_2 pour accès, etc…
v	Tous les paquets de règles du module correspondant au préfixe par défaut. *v* est unifié avec une liste de doublets de la forme *<ident, arité>*.
<i, a>	Toutes les règles composant le paquet de nom *i* et d'arité *a*.
<i, v>	Toutes les règles dont l'accès est *i*, quelque soit leur arité.
<v, a>	Toutes les règles d'arité *a* appartenant au module ayant le préfixe par défaut du contexte de lecture et écriture courant.
	Liste de doublets de l'une des trois dernières formes.

Pour les systèmes disposant d'un ou plusieurs éditeurs hôtes, on se référera à la section "Manuel d'utilisation" pour positionner la variable d'environnement (*PrologEdit*) permettant de choisir son éditeur préféré.

© PrologIA

end_module(P)

Fonction	: Déclaration de fin de module
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *P*
Messages d'erreur	: Non

Description :

Définit toutes les règles comprises entre les appels aux primitives *module* et *end_module* comme décrivant le même module. On se reportera, pour ces notions assez complexes au chapitre "Environnement". On rappelle cependant pour mémoire que *P* est le préfixe du module, et que ce préfixe doit être le même que celui défini dans la primitive *module* correspondante .

Voir également :

- *module*
- *set_context, current_context, close_context_dictionary*
- *add_implicit, remove_implicit, dictionary*

enum(R)

Fonction	: Enumération entière
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contraintes numériques
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Enumère toutes les constantes entières *n* telles que la contrainte { *R* = *n* } soit satisfaisable. L'ordre d'énumération est indéfini. L'intérêt de cette règle prédéfinie réside dans la possibilité d'énumérer des variables entières dans un domaine de valeurs limité à l'aide de contraintes numériques portant sur des rationnels ou des réels.

Exemple :

```
> enum(R),{-1.5  <  R  <  4};
  { R = 3 }
  { R = 2 }
  { R = 1 }
  { R = 0 }
  { R = -1 }
>
```


© PrologIA

enum(R, R1)

Fonction	: Enumération entière bornée supérieurement
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contraintes numériques
Paramètres connus	: *R1*
Messages d'erreur	: Non

Description :

Enumère toutes les constantes entières *n* positives telles que l'ensemble de contraintes {R = *n*, 1<=R<=R1 } soit satisfaisable. L'ordre d'énumération est indéfini.

enum(R, R1, R2)

Fonction	: Enumération entière entre deux bornes
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contraintes numériques
Paramètres connus	: *R1* et *R2*
Messages d'erreur	: Non

Description :

Enumère toutes les constantes entières *n* positives telles que l'ensemble de contraintes {*R* = *n*, 1<=*R1*<=*R*<=*R2* } soit satisfaisable. L'ordre d'énumération est indéfini.

eq(T1, T2)

Fonction	: Egalité
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose tout simplement la contrainte *{T1 = T2}*

Voir également :

- *dif*

© PrologIA

exit

Fonction	: Terminaison de Prolog III
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Permet de sortir de Prolog III en sauvant la totalité des modules, y compris ceux qui constituent le superviseur Prolog III (état initial) sous forme codée. Le fichier produit a pour nom *prolog3.psv*.

Voir également :

- *exit(S), quit, quit(N)*
- *reload*

exit(S)

Fonction	: Sortie de Prolog III
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *S*, qui doit être une chaîne de caractères
Messages d'erreur	: Chaîne attendue

Description :

Permet de sortir de Prolog III en sauvant la totalité des modules, y compris ceux qui constituent le superviseur Prolog III (état initial) sous forme codée, dans le fichier de nom S.

Voir également :

- *exit, quit, quit(N)*
- *reload*

© PrologIA

fail

Fonction	: Echec Prolog
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

fail provoque toujours un échec Prolog et donc l'exploration des autres choix en attente (backtracking). On préconise l'emploi de *fail* plutôt que l'utilisation d'un prédicat non défini, afin d'éviter d'avoir des messages d'erreurs ou d'avertissement à l'exécution; voir à ce sujet les primitives *trace* et *set_config*.

findall(V, P, L)

Fonction	: Collection de solutions
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contrôle
Paramètres connus	: *P*
Messages d'erreur	: Aucun

Description :

Unifie la liste *L* avec la collection de toutes les solutions *V* lorsqu'on efface *P*. Le but *P* doit être suffisamment connu pour être exécuté.

Exemple :

```
> list;
age(paul, 22) -> ;
age(jean, 28) -> ;
age(bruno, 25) -> ;
   {}
> findall(<N,A>,  age(N,A),  L);
   { L=[<paul,22>,<jean,28>,<bruno,25>] }
>
```


© PrologIA

find_pattern(S1, S2, N)

Fonction	: Recherche de sous-chaîne
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Arbres, listes chaînes et tuples
Paramètres connus	: *S1, S2*
Messages d'erreur	: Non

Description :

Pose la contrainte *{N = D}* où *D* est la position du début de la chaîne *S2* dans la chaîne *S1*. Si la chaîne *S2* n'est pas trouvée, alors *find_pattern* échoue.

Exemple :

```
> find_pattern("1234567890","3456",p);
  { p=3 }
> find_pattern("1234567890","abcd",p);
>
```

Voir également :

- *conc_string, substring*

free(T)

Fonction	: Vérifie que *T* ne représente pas un terme suffisamment connu
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

free(T) est exécuté avec succès si

l'on ne connaît pas l'étiquette initiale de *T*

et

l'on ne sait pas si le nombre de ses fils est nul ou non.

Echoue dans tous les autres cas.

T = E[L]

	¬ *known(L)*	*known(L)*
¬ *known(E)*	*free(T)*	?
known(E)	?	*known(T)*

Exemples :

```
> free(X);
  { X=V1[V2] }
> free((+X)[U]);
  { X !num, U !tuple }
> free(X(1,2,3));
> free(sommet[L]);
> free(sommet(X,Y,Z);
> free(sommet[L])  {L  #  <>};
>
```

Voir également :

- *free_label, known, bound, bound_tree, bound_tree'*

© PrologIA

free_label(T)

Fonction	: Vérifie que l'étiquette du terme *T* n'est pas connue
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

free_label(T) est exécuté avec succès si l'on ne connaît pas l'étiquette initiale de *T* . Echoue dans tous les autres cas.

Exemples :

```
> free_label(X);
  { X=V1[V2] }
> free_label(+X);
  { X !num }
> free_label(X(1,2,3));
  {}
> free_label(exemple[L]);
>
```

Voir également :

- *free, known, bound, bound_tree, bound_tree'*

freeze(T, P)

Fonction	: Retardement de l'exécution d'un but
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Retarde l'exécution du but *P* tant que le terme *T* n'est pas connu. On remarquera notamment que si le terme *T* est réduit à une variable numérique ou booléenne, l'ensemble de contraintes courant peut réduire suffisamment le domaine des valeurs possibles de cette variable pour provoquer l'exécution de *P.*

{ *Un terme est connu lorsque l'on connait l'étiquette initiale de l'arbre qu'il représente et que l'on sait si c'est une feuille ou non (si son nombre de fils est nul ou non nul)*

Remarque :

Lorsque *T* devient connu, les buts gelés sur *T* viennent s'insérer en tête de la résolvante. Pour des raisons d'efficacité, l'ordre dans lequel ils seront exécutés n'est pas spécifié.

Voir également :

- *known, free*

© PrologIA

garbage_collection

Fonction	: Récupération de mémoire
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Provoque l'exécution du mécanisme de récupération de mémoire. Ceci est utile dans la mesure où Prolog III gère automatiquement la mémoire qui lui est allouée à chaque étape de la résolution. Il peut ainsi s'avérer nécessaire, pour traiter une étape particulièrement coûteuse, de provoquer ce mécanisme par programme, de manière à pouvoir disposer du maximum d'espace disponible. En effet, la place occupée avant le traitement de cette étape peut n'être pas suffisamment importante pour motiver un déclenchement automatique du récupérateur de mémoire.

Voir également :

- *state*

gcd(N1, N2, N3)

Fonction	: Calcul du plus grand commun diviseur
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1, N2*
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N3'}*, où *N3'* est le plus grand commun diviseur des entiers *N1* et *N2*. Le plus grand commun diviseur calculé ne tient pas compte du signe. *gcd* échoue si *N1* et *N2* ne sont pas connus ou pas entiers.

Exemples :

```
> gcd(24,16,P);
  { P = 8 }
> gcd(24,-16,P);
  { P = 8 }
>
```

Voir également :

- *lcm*
- *pgcd, ppcm*

© PrologIA

get_config(S,V)

Fonction	: Etat courant de configuration
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *Aucun*
Messages d'erreur	: Non

Description :

Si *S* est une chaîne, pose la contrainte *{V = S'}*, où *S'* est la valeur courante associée au champ *S*. Sinon, *get_config* énumère par backtracking les champs possibles pour *S* et leur valeur courante pour *V*. Pour la liste complète de ces champs et de ces valeurs, on se reportera aux explications données à *set_config*.

Exemple :

```
> get_config("syntax",S).
  { S = "Edimbourg" }
>
```

Voir également :

- *set_config, set_configuration, get_configuration*

get_configuration(U)

Fonction	: Etat courant des configurations
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{U = U'}*, où *U'* est un tuple de doublets *<champ, valeur>* qui décrit l'état de tous les paramètres courants. Pour la liste des champs et de leurs valeurs possibles, voir *set_config*.

Exemple :

```
> get_configuration(X).
  { X=<<"echo",0>,<"epsilon_float",7.450580596923828e-09>,
    <"format_decimal",3>,<"format_float","%.15le">,
    <"format_out_num","Fnormal">,<"statistics",0>,
    <"syntax","Edimbourg">,<"trace",0>,<"tty_wrap",200>,
    <"undefined_rule","fail">> }
>
```

Voir également :

- *set_config, set_configuration, get_config*

© PrologIA

getenv(S1,S2)

Fonction	: Obtenir la valeur d'une variable de l'environnement
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *S1*
Messages d'erreur	: Non

Description :

Pose la contrainte *{S2 = S2'},* où *S2'* est la valeur associée au sein du système d'exploitation à la variable d'environnement de nom *S1.* Par exemple, sous UNIX une telle association pourra se faire par le biais de la commande shell *setenv* .

Exemple :

```
$ setenv essai 17
$ prolog3
.....
> getenv("essai",S);
  { S = "17" }
>
```

ident(I)

Fonction	: Vérifie que *I* représente un identificateur connu
Classe	: Procédure externe Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *I* représente un identificateur connu, et échoue dans le cas inverse.

Exemples :

```
> ident(nuit_blanche);
  {}
> ident("nuit_blanche");
>
  ident('*');
  {}
> ident(`*`);
>
```

Remarques :

Dans le troisième exemple, le but est exécuté avec succès, car il s'agit d'un identificateur quoté. Dans le quatrième, c'est un caractère, et donc un échec.

Voir également :

- *is_ident*

© PrologIA

in_char(C)

Fonction	: Lecture d'un caractère
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit un caractère sur l'unité active d'entrée. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un close_input.

Exemples :

```
> in_char(X)  in_char(Y);
a
  { X=`\12`, Y=`a` }
> in_char(X);1
  { X=`1` }
>
```

Voir également :

- *in_char', in_ident, in_integer, inl, in_real, in_sentence, in_string, in_term, next_char, next_char'*

in_char'(C)

Fonction	:	Lecture d'un caractère qui ne soit pas un caractère de formatage
Classe	:	Procédure externe Prolog III (II+)
Catégorie	:	Entrées/sorties
Paramètres connus	:	Aucun
Messages d'erreur	:	Fin de fichier inattendue

Description :

Lit le premier caractère dont le code ASCII n'est pas inférieur ou égal à 32 (caractères de formatage, espace, retour chariot, tabulations, etc…) sur l'unité active d'entrée. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Exemples :

```
> in_char'(C);
a
  { C = `a` }
> in_char'(C);


                a
  { C = `a` }
> in_char'(C)  in_char(C');
a
  { C = `a`, C' = `\15` }
>
```

Voir également :

- *in_char, in_ident, in_integer, inl, in_real, in_sentence, in_string, in_term, next_char, next_char'*

© PrologIA

in_ident(I)

Fonction	: Lecture d'un identificateur
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit un identificateur sur l'unité active d'entrée. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Exemples :

```
> in_ident(I);
ceci_est_un_identificateur
  { I = ceci_est_un_identificateur }
> in_ident(I);
'->'
  { I = '->' }
> in_ident(I);
[]
>
  in_ident(I);
V_ariable
>
```

Voir également :

- *in_char, in_char', in_integer, inl, in_real, in_sentence, in_string, in_term, next_char, next_char'*

in_integer(N)

Fonction	: Lecture d'un entier
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit un entier sur l'unité active d'entrée. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Exemples :

```
> in_integer(N);
124578956810000
  { N = 124578956810000 }
> in_integer(N);
ceci  n'est  manifestement  pas  un  entier
>
  in_integer(N)  in_ident(I);
123nous_irons_au_bois
  { N = 123, I = nous_irons_au_bois }
> in_integer(N)  in_real(R);
1213.06
  { N = 1213, R = 6.000000000000000e-02 }
>
```

Remarque :
in_integer lit un objet ayant la *syntaxe* d'un entier. Par exemple `12/2` ne sera pas lu comme l'entier `6`, seul l'entier `12` sera pris en compte par *in_integer*.

Voir également :

- *in_char, in_char', in_ident, inl, in_real, in_sentence, in_string, in_term, next_char, next_char'*

© PrologIA

in_real(R)

Fonction	: Lecture d'un flottant
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit un flottant sur l'unité active d'entrée. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Exemples :

```
> in_real(R);
52.4512789526452l3E17
  { R = 5.245127895264521e+18 }
> in_real(R);
100.00
  { R = 1.000000000000000e+02 }
> in_real(R);
100
>
```

Remarque :
in_real lit un objet ayant la *syntaxe* d'un flottant, à l'exclusion des entiers (i.e. le point décimal est obligatoire).

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_sentence, in_string, in_term, next_char, next_char'*

in_sentence(T1,T2,T3)

Fonction	: Lecture d'une phrase
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit une phrase qui se termine par ".", "?" ou "!" et la met sous la forme de trois listes, respectivement unifiées avec T_1, T_2 et T_3. T_1 est la liste des unités lexicales (mots, nombres, caractères spéciaux, etc…) constituant la phrase ; T_3 est la liste obtenue en remplaçant chaque lettre majuscule apparaissant dans un élément de T_1 par la lettre minuscule correspondante. T_2 est la même chose que T_3, mais chaque élément de T_3 qui correspond à un identificateur connu dans la famille du préfixe par défaut du contexte courant a été remplacé par le symbole conventionnel <> pour indiquer ce fait.

Exemple :

```
> fleuve(Rhone)->;
> in_sentence(T1,T2,T3);
La vie est un long fleuve tranquille.
   { T1 = <"La","vie","est","un","long","fleuve","tranquille",".">,
     T2 = <"la","vie","est","un","long",<>,"tranquille",".">,
     T3 = <"la","vie","est","un","long","fleuve","tranquille","."> }
>
```

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_string, in_term, next_char, next_char'*

© PrologIA

in_sentence'(T)

Fonction	: Lecture d'une phrase
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit une phrase qui se termine par ".", "?" ou "!" et la met sous la forme d'une liste unifiée avec *T*. La liste*T* contient les unités lexicales (mots, nombres, caractères spéciaux, etc…) constituant la phrase.

Exemple :

```
> fleuve(Rhone)->;
> in_sentence(T);
La vie est un long fleuve tranquille.
  { T1 = <"La","vie","est","un","long","fleuve","tranquille","."> }
>
```

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_string, in_term, next_char, next_char'*

in_string(S)

Fonction	: Lecture d'une chaîne quelconque
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lecture d'une chaîne de caractères sur l'unité d'entrée courante. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Exemples :

```
> in_string(S);
"ceci  est  une  chaine"
  { S = "ceci est une chaine" }
> in_string(S);
ceci  n'est  pas  une  chaine
>
  in_string(S);
"126/45"
  { S = "126/45" }
> in_string(S);
"123\
456"
  { S = "123456" }
>
```

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_sentence, in_term, next_char, next_char'*

© PrologIA

in_term(T)

Fonction	: Lecture d'un terme quelconque
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Erreur dans assert ou in_term Fin de fichier inattendue

Description :

Lecture d'un terme Prolog sur l'entrée courante.

Lit la plus grande suite *x* de caractères qui constituent un terme. Le terme doit se terminer par un point-virgule, qui n'est pas lu.

Exemples :

```
> in_term(X);
arbre(-3/4,<a,b>)  in_char(Y);
  { X=arbre(-3/4,<a,b>), Y=`;` }
> in_term(X);
terme1(u)  terme2(v), { u < v }
  { X=terme1(u), u !num }
>
```

Remarques :

1. Eviter d'inclure des espaces dans tout ce qui n'est pas parenthésé.
2. Seul le premier terme est pris en compte.
3. Il peut y avoir un système de contraintes qui, lui, est pris en compte.

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_sentence, in_string, next_char, next_char'*

inl(S)

Fonction	: Lecture dune ligne
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue

Description :

Lit une ligne jusqu'au premier retour chariot (\n) sur l'unité active d'entrée. Le résultat est donné dans la chaine de caractères *S*. Si une fin de fichier est détectée, une erreur est signalée et tout se passe comme si on avait fait un *close_input*.

Voir également :

- *in_char, in_char', in_ident, in_integer in_real, in_sentence, in_string, in_term, next_char, next_char'*

© PrologIA

input(F)

Fonction	: Définition de l'unité d'entrée courante
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *F*, qui doit être une chaîne de caractères
Messages d'erreur	: Fin de fichier inattendue
	Impossible d'ouvrir le fichier

Description :

L'unité dont le nom est *F* devient l'unité d'entrée courante. Si cette unité ne figure pas parmi les unités ouvertes, alors le fichier correspondant est recherché et ouvert. Sinon, cette unité passe au sommet de la pile, sauf si elle s'y trouvait déjà, auquel cas le message « Fichier déjà ouvert » est produit.

Un fichier de commandes ouvert par *input* peut lui-même contenir une commande *input*, la nouvelle unité courante est alors empilée au-dessus de la précédente.

Voir également :

- *input_is, output, output_is*

input_is(F)

Fonction	: Fournit l'unité d'entrée courante
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier inattendue
	Impossible d'ouvrir le fichier

Description :

Pose la contrainte **{F = S}**, ou **S** est une chaîne de caractères décrivant l'unité d'entrée courante.

Exemples :

```
> input_is(X);
  { X = "console" }
>
```

Voir également :

- *input, output, output_is*

© PrologIA

insert

Fonction	: Insertion de règles à partir de l'unité courante
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Erreur de syntaxe. Paquet déjà existant
	Fin de fichier inattendue

Description :

Cette règle prédéfinie fait basculer le système dans un mode pour lequel les règles lues sur l'unité d'entrée courante sont ajoutées au programme en cours, dans l'ordre dans lequel elles sont lues. Les requêtes rencontrées sont ignorées. Le mode *insert* se termine soit quand une règle vide est trouvée, soit si l'on rencontre la fin du fichier d'entrée.

Exemple :

```
> insert;
conc([],y,y)  ->;
conc([e|x],y,[e|z])  ->  conc(x,y,z);;
>
```

Si une erreur de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un « ; », ou jusqu'à la fin de la ligne sur l'unité *console*) sont ignorés.
insert provoque une erreur lorsque un paquet lu existe déjà, alors que *reinsert* remplace l'ancien paquet par la nouvelle définition.

Voir également :

- *insert(F), reinsert, reinsert(F)*

insert(F)

Fonction	: Insertion de règles à partir d'un fichier
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *F*, qui doit être une chaîne.
Messages d'erreur	: Fin de fichier inattendue.
	Impossible d'ouvrir le fichier
	Erreur de syntaxe. Paquet déjà existant

Description :

Cette règle prédéfinie permet d'ajouter des règles, dans l'ordre dans lequel elles sont lues à partir du fichier *F*. Les requêtes rencontrées sont ignorées. La lecture se termine soit lorsque l'on rencontre la fin du fichier, soit lorsqu'on rencontre une règle vide, c'est-à-dire (en syntaxe de base) ne contenant qu'un « ; » .

Si une erreur de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un «;») sont ignorés, puis l'insertion reprend.

insert provoque une erreur lorsque un paquet lu existe déjà. Dans ce cas, Prolog III sort du « mode insertion », mais ne ferme pas les fichiers qui étaient ouverts : il continue donc à lire le(s) fichier(s) en cours : les règles sont insérées et les requêtes exécutées.

Selon que l'on a par ailleurs activé ou non le mode écho (cf. règle *echo*) les règles sont affichées sur la console au fur et à mesure de leur lecture.

Voir également :

- *insert, reinsert, reinsert(F)*

© PrologIA

integer(N)

Fonction	: Vérifie que *I* est un nombre entier connu
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Vérifications de types
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

S'exécute avec succès si *N* représente un nombre entier connu. Echoue dans tous les autres cas.

Exemples :

```
> integer(0);
  {}
> integer(-1234*4567);
  {}
> integer(X+2);
>
```

Voir également :

- *num, is_num*

is_bool(B)

Fonction	: Vérifie que *B* représente une valeur booléenne.
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *B* représente une valeur booléenne, c'est à dire que la contrainte *{b !bool}* appartient au système de contraintes courant. Echoue dans le cas contraire.

Exemples :

```
> is_bool(1');
  {}
> is_bool(B),  {B  !bool};
  { B !bool }
> is_bool(B),  {B  !boolt};
>
```

Voir également :

- *bool*

© PrologIA

is_char(C)

Fonction	: Vérifie que *C* représente un caractère
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *C* représente un caractère, c'est à dire que la contrainte *{C !char}* appartient au système courant. Echoue dans le cas contraire.

Exemples :

```
> is_char(`$`);
  {}
> is_char(C)  {C  !char};
  { C !char }
> is_char('$')
>
```

Remarques :

On rappelle que les caractères sont notés entre anti-quotes (voir exemples 1 ci-dessus). C'est la raison pour laquelle l'exemple 3 échoue ('$' est un identificateur quoté). Pour vérifier que *C* est un caractère connu, utiliser *char(C).*

Voir également :

- *char, string, is_string*

is_ident(I)

Fonction	: Vérifie que *I* représente un identificateur
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *I* représente un identificateur connu, c'est à dire que la contrainte *{I !id}* appartient au système courant. Echoue dans le cas contraire.

Exemples :

```
> is_ident(gwendoline);
  {}
> is_ident('ceci  en  est  un  aussi');
  {}
> is_ident([]);
  {}
> is_ident(I)  {I  !idt};
>
```

Remarques :

Dans le second exemple, *'ceci en est un aussi'*, est un identificateur quoté, et le but est exécuté correctement.

Voir également :

- *ident*

© PrologIA

is_leaf(T)

Fonction	: Vérifie que *T* représente une feuille
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *T* représente un arbre réduit à une feuille, c'est à dire que la contrainte *{T :: 0}* appartient au système courant. Echoue dans le cas contraire.

Exemples :

```
> is_leaf(X[<>]);
  { X::0 }
> is_leaf(B1|B2);
  { B1 !bool, B2 !bool }
> is_leaf(U.V)  {U::2};
>
```

Voir également :

- *bound_tree, bound_tree'*

is_num(R)

Fonction	: Vérifie que *R* représente un nombre
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *R* représente une valeur numérique, c'est à dire que la contrainte *{ R !num }* appartient au système courant. Echoue dans le cas contraire.

Exemples :

```
> is_num(91252525);
  {}
> is_num(3.12e12*X);
  { X !num }
> is_num(X);
>
```

Voir également :

- *num, integer*

© PrologIA

is_tuple(U)

Fonction	: Vérifie que *U* représente un tuple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que le terme *U* représente un tuple, c'est à dire que la contrainte *{ U !tuple }* appartient au système courant. Echoue dans le cas contraire.

Exemples :

```
> is_tuple(<A,B,C>);
  {}
> is_tuple(U),{X  =  E[U]};
  { X = E[U] }
> is_tuple("une  chaine  cache  un  tuple  !");
  {}
> is_tuple([A,B,C]);
>
```

Voir également :

- *tuple*

is_univ(T)

Fonction	: Vérifie que le terme *T* représente un arbre non typé
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Réussit si le terme *T* n'est pas typé caractère, identificateur, numérique, booléen, tuple ou liste binaire.

Exemples :

```
> is_univ(X);
  {}
> is_univ(E[U]);
  { E :: 0,
    U !tuple }
> is_univ(arbre(1,A));
  {}
> is_univ((+X)(A,B,C));
  { X !num }
> is_univ(ident);
>
  is_univ([1,2,3]);
>
  is_univ(2X+4);
>
  is_univ(B) { B !bool };
>
```


© PrologIA

kill_module(S)

Fonction	: Suppression des règles d'un module
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *S*
Messages d'erreur	: Pas d'identificateur avec ce préfixe

Description :

Supprime toutes les règles du module désigné par la chaîne de caractères *S*. Les tableaux du module sont désalloués, et les assignations des identificateurs du module sont défaites.

Exemple :

```
> essai1  ->;
> essai2  ->;
> assign(essai3,0);
  {}
> list  val(essai3,X);
  essai1 ->  ;
  essai2 ->  ;
  { X = 0 }
> kill_module("");
  {}
> list  val(essai3,X);
  { X = essai3 }
>
```

Voir également :

- *new*

known(T)

Fonction	: Vérifie que *T* représente un terme connu
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *T*
Messages d'erreur	: Non

Description :

S'exécute avec succès si le terme *T* représente un terme suffisamment connu, c'est à dire un arbre :

- dont l'étiquette est connue
- dont on sait si le nombre de fils est nul ou non

Echoue dans le cas contraire.

T = E[L]

	¬ *known(L)*	*known(L)*
¬ *known(E)*	*free(T)*	?
known(E)	?	*known(T)*

Exemples :

```
> known(E[<>]);
>
  known(arbre[U]);
>
  known(arbre[U]),  {U#<>};
  { U=X1.X2, X1::1 }
> known(+X);
>
```

Voir également :

- *free, bound*

© PrologIA

known_part(U1, U2, U3)

Fonction	: Extraction de la partie connue d'un tuple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Pose la contrainte *{U1 = U2.U3},* où *U2* est un tuple formé de la plus grande suite connue des premiers éléments de *U1*. Par « connu » on entend : « avoir une étiquette connue » et « savoir si le nombre de fils est nul ou pas »

Exemples :

```
> known_part(U1,U2,U3),
  {U1  =<1,2,3>.<4,X,6>};
  { U1 = <1,2,3,4,X,6>,
  U2 = <1,2,3,4,X,6>,
  U3 = <> }
> known_part(U1,U2,U3),
  {U1  =<1,2,3>.<X>.L};
  { U1 = <1,2,3,X>.L,
  U2 = <1,2,3,X>,
  U3 = L }
> known_part(U1,U2,U3),
  {U1  =<1,2,3>.<X>.L,L#<>};
  { U1 = <1,2,3,X>.L,
  U2 = <1,2,3,X>.V1,
  U3 = V2,
  L = V1.V2,
  V1::1 }
>
```

lcm(N1, N2, N3)

Fonction	: Calcul du plus petit commun multiple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1, N2*
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N3'}*, où *N3'* est le plus petit commun multiple des entiers *N1* et *N2*. *lcm* échoue si *N1* et *N2* ne sont pas connus ou pas entiers.

Exemples :

```
> lcm(12,18,P);
  { P = 36 }
> lcm(12,-18,P);
  { P = -36 }
>
```

Voir également :

- *ppcm*
- *pgcd, gcd*

© PrologIA

line

Fonction	: Saut de ligne
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit un retour chariot sur l'unité de sortie courante

Exemple :

```
> line;

  {}
>
```

Remarques :

Cette procédure externe pourra peut-être se passer de remarques.

Voir également :

- *outl, outml*

list

Fonction	: Impression de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Liste sur l'unité courante de sortie toutes les règles du module déterminé par le préfixe par défaut du contexte courant.

Remarque :

Les règles sont d'abord décodées, puis ensuite listées sur l'unité courante de sortie. Elles peuvent donc avoir une forme qui diffère légèrement de leur forme d'origine.

© PrologIA

list(T)

Fonction	: Impression de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Argument de mauvais type, Règle inexistante

Description :

Liste sur la sortie courante le ou les paquets de règles indiqués par *T* qui doit être un terme de la forme indiquée ci-dessous (*s* désigne une chaîne de caractères, *i* un identificateur, et *v* une variable) :

s Tous les paquets de règles du module correspondant au préfixe *s*

i Le(s) paquet(s) de règles ayant *i* pour accès.

[i_1, i_2, … i_n]
Tous le(s) paquet(s) de règles ayant i_1 pour accès puis le(s) paquet(s) ayant i_2 pour accès, etc…

v Tous les paquets de règles du module correspondant au préfixe par défaut. *v* est unifié avec une liste de doublets de la forme *<ident, arité>*.

list(<I, A>)

Fonction	: Impression de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Argument de mauvais type, Règle inexistante

Description :

Liste le(s) paquet(s) de règles de nom *I* et d'arité *A* du module déterminé par le préfixe par défaut du contexte courant (*V* désigne une variable).

- *<I, A>* Toutes les règles composant le paquet de nom *i* et d'arité *A*.
- *<I, V>* Toutes les règles dont l'accès est *I*, quelque soit leur arité.
- *<V, A>* Toutes les règles d'arité *A* appartenant au module ayant le préfixe par défaut du contexte de lecture et écriture courant dont l'arité est *A*.

© PrologIA

list([<I1, A1>, …, <In, An>])

Fonction	: Impression de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Argument de mauvais type, Règle inexistante

Description :

Liste chacun des paquets de règles désignés par les éléments de la liste de la même façon que *list(<A,I>)*.

list(<I, A>,N)

Fonction	: Impression de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Argument de mauvais type, Règle inexistante

Description :

Liste sur l'unité courante de sortie la règle numéro *N* du paquet correspondant au nom *I* et à l'arité (nombre d'arguments) *A*. Chacun des paramètres *I, A,* ou *N* peut être représenté par une variable.

© PrologIA

list_string(L, S)

Fonction	: Conversion liste/chaîne
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: *L*
Messages d'erreur	: Non

Description :

Pose la contrainte *{S = L'}*, où *L'* est une chaîne composée des caractères qui forment la liste *L*.

Exemple :

```
> list_string([`H`,`e`,`l`,`l`,`o`],  X);
  { X = "Hello" }
>
```

Voir également :

- *split*

list_tuple(L, U)

Fonction	: Conversion liste/tuple
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: *L*
Messages d'erreur	: Non

Description :

Pose la contrainte *{S = U'},* où *U'* est un tuple composée des éléments qui forment la liste *L.*

Exemple :

```
> list_tuple([1/12,aaa,1',"rhino"],  X);
  { X = <1/12,aaa,1',"rhino"> }
> list_tuple([1,[2,[3]]],  X);
  { X = <1,[2,[3]]> }
>
```

Voir également :

- *split*

© PrologIA

lower_bound(R1, R2)

Fonction	: Borne inférieure d'une expression numérique
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R1*
Messages d'erreur	: Expression non bornée (Erreur 59) Arguments de mauvais type (Erreur 60)

Description :

Calcule la borne inférieure *m* (i.e. le plus grand des minorants) de *R1* et ajoute au système de contraintes courant la contrainte *R2 = m* .

Provoque une erreur lorsque *R1* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R1* n'est pas bornée inférieurement (Erreur 59).

Exemples :

```
> lower_bound(R,R'),{R  =  2X+Y,  X>=2,  Y>3};
  { R' = 7, R = ... }
> lower_bound(R,R'),{R  =  2X+Y,  X>=2};
Err 59 : Expression non bornee lors d'un calcul d'extremum
> lower_bound(R,R),{R  =  2X+Y,  X>=2,  Y>3};
>
```

Remarques :

On rappelle que, dans l'ensemble des réels, toute partie non vide qui possède un minorant admet une borne inférieure. On veillera donc au préalable à borner *R1* inférieurement.

On notera également que l'exécution de *lower_bound(R1, R2)* ne pose pas la contrainte *R1 = R2.*

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *upper_bound, maximum, minimum, maximize, minimize*

upper_bound(R1, R2)

Fonction	: Borne supérieure d'une expression numérique
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R1*
Messages d'erreur	: Expression non bornée (Erreur 59) Arguments de mauvais type (Erreur 60)

Description :

Calcule la borne supérieure *m* (i.e. le plus petit des majorants) de *R1* et ajoute au système de contraintes courant la contrainte *R2 = m* .

Provoque une erreur lorsque *R1* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R1* n'est pas bornée supérieurement (Erreur 59).

Exemples :

```
> upper_bound(R,R'),{R  =  2X+Y,  X<=2,  Y<3};
  { R' = 7, R = ... }
> upper_bound(R,R'),{R  =  2X+Y,  X<=2};
Err 59 : Expression non bornee lors d'un calcul d'extremum
> upper_bound(R,R),{R  =  2X+Y,  X<=2,  Y<3};
>
```

Remarques :

On rappelle que, dans l'ensemble des réels, toute partie non vide qui possède un majorant admet une borne supérieure. On veillera donc au préalable à borner *R1* supérieurement.

On notera également que l'exécution de *upper_bound(R1, R2)* ne pose pas la contrainte *R1 = R2.*

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *lower_bound, maximum, minimum, maximize, minimize*

© PrologIA

maximum(R1, R2)

Fonction	: Maximum d'une expression numérique
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R1*
Messages d'erreur	: Expression non bornée (Erreur 59) Arguments de mauvais type (Erreur 60)

Description :

Calcule le maximum *m* de *R1* et ajoute au système de contraintes courant la contrainte *R2 = m* .

Echoue lorsque *R1* est borné supérieurement sans pour autant avoir un maximum (la borne supérieure ne peut être atteinte par *R1*).

Provoque une erreur lorsque *R1* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R1* n'est pas bornée supérieurement (Erreur 59).

Exemples :

```
> maximum(R,R'),{R  =  2X+Y,  X<=2,  Y<=3};
  { R' = 7, R = ... }
> maximum(R,R'),{R  =  2X+Y,  X<=2};
Err 59 : Expression non bornee lors d'un calcul d'extremum
> maximum(R,R'),{R  =  2X+Y,  X<=2,  Y<3};
>
```

Remarque :

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *minimum, lower_bound, upper_bound, maximize, minimize*

maximize(R)

Fonction	:	Maximiser une expression numérique
Classe	:	Règle prédéfinie Prolog III
Catégorie	:	Contraintes numériques
Paramètres connus	:	*R*
Messages d'erreur	:	Expression non bornée (Erreur 59) Argument de mauvais type (Erreur 60)

Description :

Equivalent (mais plus performant) à *maximum(R, R)*

Calcule le maximum *m* de *R* et ajoute au système de contraintes courant la contrainte *R = m* .

Echoue lorsque *R* est borné supérieurement sans que le maximum puisse être atteint par *R*.

Provoque une erreur lorsque *R* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R* n'est pas bornée supérieurement (Erreur 59).

Exemples :

```
> maximize(R),{R  =  2X+Y,  X<=2,  Y<=3};
  { R = 7 , X = 2 , Y = 3 }
> maximize(R);
Err 59 : Expression non bornee lors d'un calcul d'extremum
> maximize(R),{R  !numt,  R::2};
Err 60 : Argument de mauvais type pour un calcul d'extremum
> maximize(R),{R  =  2X+Y,  X<=2,  Y<3};
>
```

Remarque :

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *minimize, maximum, minimum, lower_bound, upper_bound*

© PrologIA

minimum(R1, R2)

Fonction	: Minimum d'une expression numérique
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R1*
Messages d'erreur	: Expression non bornée (Erreur 59) Arguments de mauvais type (Erreur 60)

Description :

Calcule le minimum *m* de *R1* et ajoute au système de contraintes courant la contrainte *R2 = m* .

Echoue lorsque *R1* est borné inférieurement sans pour autant avoir un minimum (la borne inférieure ne peut être atteinte par *R1*).

Provoque une erreur lorsque *R1* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R1* n'est pas bornée inférieurement (Erreur 59).

Exemples :

```
> minimum(R,R'),{R  =  2X+Y,  X>=2,  Y>=3};
  { R' = 7, R = ... }
> maximum(R,R'),{R  =  2X+Y,  X>=2};
Err 59 : Expression non bornee lors d'un calcul d'extremum
> maximum(R,R'),{R  =  2X+Y,  X>=2,  Y>3};
>
```

Remarque :

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *maximum, lower_bound, upper_bound, maximize, minimize*

minimize(R)

Fonction	:	Minimiser une expression numérique
Classe	:	Règle prédéfinie Prolog III
Catégorie	:	Contraintes numériques
Paramètres connus	:	*R*
Messages d'erreur	:	Expression non bornée (Erreur 59) Argument de mauvais type (Erreur 60)

Description :

Equivalent (mais en plus performant) à *minimum(R, R)* .

Calcule le minimum *m* de *R* et ajoute au système de contraintes courant la contrainte *R = m* .

Echoue lorsque *R* est borné inférieurement sans que le minimum puisse être atteint par *R*.

Provoque une erreur lorsque *R* ne peut être typé numérique (Erreur 60) ou encore lorsque l'expression numérique *R* n'est pas bornée inférieurement (Erreur 59).

Exemples :

```
> minimize(R),{R  =  2X+Y,  X>=2,  Y>=3};
  { R = 7 , X = 2 , Y = 3 }
> minimize(R);
Err 59 : Expression non bornee lors d'un calcul d'extremum
> minimize(R),{R  !numt,  R::2};
Err 60 : Argument de mauvais type pour un calcul d'extremum
> minimize(R),{R  =  2X+Y,  X>=2,  Y>3};
>
```

Remarque :

On pourra utiliser la primitive *block* pour la récupération des erreurs.

Voir également :

- *maximize, maximum, minimum, lower_bound, upper_bound*

© PrologIA

mod(N1, N2, N3)

Fonction	: Reste de la division entière
Classe	: Procédure externe Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1* et *N2 entier, N2 non nul*
Messages d'erreur	: Division par zéro

Description :

Pose la contrainte *{N3 = R}*, où *R* est le reste de la division entière de la valeur absolue de *N1* par la valeur absolue de *N2*. On prendra garde à ne pas confondre le prédicat avec la fonction évaluable du même nom (voir la règle prédéfinie *val).*

Exemples :

```
> mod(9,2,R);
  { N = 1 }
> mod(7,N1,N2);
>
```

Voir également :

- *div*
- *trunc*

module(P1, S1, S2, P2)

Fonction	: Déclaration de début de module
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Définit toutes les règles comprises entre les appels des primitives *module* et *end_module* comme décrivant un même module. On se reportera, pour ces notions assez complexes au chapitre "Environnement". On rappelle cependant pour mémoire que *P1* est le préfixe du module, *S1*, la suite explicite, *S2*, la suite implicite, et *P2* le préfixe par défaut.

On peut utiliser module avec seulement le premier, les deux premiers ou les trois premiers arguments dans les conditions décrites précisément au chapitre "Environnement" :

module(P)	est équivalent à	*module(P, [], ["sys"], P)*
module(P, S)	est équivalent à	*module(P, S, ["sys"], P)*
module(P, S, I)	est équivalent à	*module(P, S, I, P)*

Voir également :

- *end_module*
- *set_context, current_context, close_context_dictionary*
- *add_implicit, remove_implicit, dictionary*

© PrologIA

mult(N1, N2, N3)

Fonction	: Multiplication avec retardement
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N1*N2}* avec, s'il y a lieu, les retardements nécessaires.

Exemples :

```
> mult(X,Y,Z);
  { Z = X * Y }
>
```

Voir également :

- *bound_mult*

new

Fonction	: Suppression des règles d'un module
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Supprime toutes les règles du module déterminé par le préfixe par défaut du contexte courant. Contrairement à *kill_module* les tableaux du module ne sont pas désalloués, et les assignations des identificateurs du module ne sont pas défaites.

Exemple :

```
> essai1  ->;
> essai2  ->;
> assign(essai3,10);
  {}
> list  val(essai3,X);
  essai1 ->  ;
  essai2 ->  ;
  { X = 10 }
> new;
  {}
> list  val(essai3,X);
  { X = 10 }
>
```

Voir également :

- *kill_module*

© PrologIA

next_char(C)

Fonction	: Lecture anticipée d'un caractère
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrée/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{C = C'}*, où *C'* est le prochain caractère rencontré sur l'unité d'entrée courante sans toutefois le lire effectivement.

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_sentence, in_string, in_term, next_char'*

next_char'(C)

Fonction	: Lecture anticipée d'un caractère qui ne soit pas un caractère de formatage
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrée/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{C = C'}*, où *C'* est le prochain caractère rencontré qui ne soit pas un caractère de formatage sur l'unité d'entrée courante sans toutefois le lire effectivement. En fait, tous les caractères dont le code ASCII est inférieur ou égal à 32 sont ignorés.

Voir également :

- *in_char, in_char', in_ident, in_integer, inl, in_real, in_sentence, in_string, in_term, next_char*

© PrologIA

no_debug

Fonction	: Annule le mode debug
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Annule l'effet des primitives *debug* ou de tout passage en mode `debug` provoqué par l'utilisation de la primitive *set_config*.

Voir également :

- *debug, trace, no_trace,*
- *set_config, set_configuration, get_config, get_configuration*

no_echo

Fonction	: Annule l'écho sur la console
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Annule l'effet de la primitive *echo.*

Voir également :

- *echo*
- *set_config, set_configuration, get_config, get_configuration*

© PrologIA

no_spy(<I, N>)
no_spy(S)

Fonction	: Annule l'espionnage de paquets de règles
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *S*, au moins un parmi *I* et *N*
Message d'erreur	: Module inexistant

Description :

Enlève tout point-espion installé sur un ou plusieurs paquets de règles.

Dans la première forme, *I* désigne l'identificateur de paquet et *N* l'arité de ce paquet, et au moins l'une des deux informations doit être connue à l'appel. Un échec survient dans le cas contraire.

Dans la deuxième forme, *S* est un nom de module sous la forme d'une chaîne de caractère. Toutes les règles du module *S* sont alors concernées. Provoque une erreur s'il n'existe aucune règle de préfixe *S*.

L'utilisateur se réfèrera au manuel “Mise au point de programmes Prolog III” pour de plus amples informations.

Voir également :

- *spy*
- *debug, no_debug*

no_trace

Fonction	: Annule la trace
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Annule l'effet des primitives *trace* ou de toute trace provoquée par l'utilisation de la primitive *set_config*. Revient au mode normal d'exécution d'un programme Prolog III.

Voir également :

- *trace, debug, no_debug, suspend_trace, resume_trace*
- *set_config, set_configuration, get_config, get_configuration*

© PrologIA

not(T)

Fonction	: Négation par échec
Classe	: Règle prédéfinie Prolog III (Edinburgh)
Catégorie	: Contrôle
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Réalise une négation par échec. Voici la règle *not(T) in extenso* :

```
not(P) -> P ! fail;
not(P) ->;
```

Et en syntaxe Edinburgh :

```
not(P) :- P, !, fail .
not(P) .
```

Remarque :

Puisque *not* lance un but, son comportement peut dépendre du positionnement du paramètre *"undefined_rule"* de la primitive *set_config* , si on est susceptible de lancer des buts non-définis.

num(R)

Fonction	: vérifie que R représente un nombre connu
Classe	: Règle prédéfinie Prolog III
Catégorie	: Vérification de types
Paramètres connus	: *R*
Messages d'erreur	: Non

Description :

Cette règle prédéfinie vérifie que *R* est un terme représentant une valeur numérique connue. Si ce n'est pas le cas *num(R)* échoue.

Exemples :

```
> num(1960);
  {}
> num(3X+2);
>
  num(X),{X>0,Y>0,X+Y=12,2X+4Y=34};
  { X = 7, Y = 5 }
>
```

Remarques :

Cette règle prédéfinie ne teste pas seulement que *R* représente une valeur numérique (la règle prédéfinie *is_num(R)* effectue cette vérification).

Le terme *R* peut être contraint à représenter une valeur numérique connue dans le cas où, pour toutes les solutions du système de contraintes courant, *R* représente la même valeur (voir le troisième exemple ci-dessus).

© PrologIA

numden(R, N1, N2)

Fonction	: Calcul du numérateur et du dénominateur d'un rationnel
Classe	: Procédure externe Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R*
Messages d'erreur	: Non

Description :

Pose les contraintes *{N1 = N1', N2 = N2'},* où *N1'* est le numérateur de *R,* et *N2'* le dénominateur de *R.* Echoue s*i R* n'est pas connu.

Exemples :

```
> numden(17,N1,N2);
  { N1 = 17,N2 = 1 }
> numden(9/6,N1,N2);
  { N1 = 3,N2 = 2 }
> numden(R,14,2);
>
```

out(T)

Fonction	: Ecriture de termes
Classe	: Règle prédéfinie Prolog III
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit le terme *T* sur l'unité de sortie courante, sans les contraintes associées aux variables qui peuvent figurer dans ce terme.

Voir également :

- *outc, outl, outm, outml*

© PrologIA

outc(T)

Fonction	: Ecriture de termes avec contraintes
Classe	: Règle prédéfinie Prolog III
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit le terme *T* sur l'unité de sortie courante, avec les contraintes associées aux variables qui peuvent figurer dans ce terme.

Voir également :

- *out, outl, outm, outml*

outl(T)

Fonction	: Ecriture de termes avec <CR>
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit le terme *T* sur l'unité de sortie courante, sans les contraintes associées aux variables qui peuvent figurer dans ce terme, suivis d'un retour-chariot.

Voir également :

- *out, outc, outm, outml*

© PrologIA

outm(S1, S2, …, SN)

Fonction	: Ecriture de chaînes
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *S1, S2, …, SN*
Messages d'erreur	: Argument de mauvais type

Description :

Ecrit les chaînes *S1, S2, …, SN* (N ≥ 1) sur l'unité de sortie courante. Une erreur est signalée si une de ces chaînes n'est pas connue.

Voir également :

- *out, outc, outl, outml*

outml(S1, S2, …, SN)

Fonction	: Ecriture de chaînes avec <CR>
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *S1, S2, …, SN*
Messages d'erreur	: Argument de mauvais type

Description :

Ecrit les chaînes *S1, S2, …, SN* (N ≥ 1) sur l'unité de sortie courante, suivies d'un retour chariot. Une erreur est signalée si une de ces chaînes n'est pas connue.

Voir également :

- *out, outc, outl, outm*

© PrologIA

output(F)

Fonction	: Définition de l'unité de sortie courante
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *F*, qui doit être une chaîne.
Messages d'erreur	: Non

Description :

L'unité dont le nom est *F* devient l'unité de sortie courante. Si cette unité ne figure pas parmi les unités ouvertes, alors un fichier ayant le nom *F* est crée. Sinon, cette unité passe au sommet de la pile, sauf si elle s'y trouvait déjà.

Voir également :

- *output_is, close_output, input*

output_is(F)

Fonction	: Fournit l'unité de sortie courante
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{F = S}*, ou *S* est une chaîne de caractères décrivant l'unité de sortie courante.

Exemples :

```
> output_is(X);
  { X = "console" }
>
```

Voir également :

- *input, input_is, output*

© PrologIA

particular_value(R1, R2)

Fonction	: Recherche d'une solution particulière
Classe	: Procédure externe Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{R2 = R1'},* où *R1'* est une valeur quelconque du terme *R1* qui satisfait l'ensemble de contraintes courant.

Exemples :

```
> particular_value(X,Y)   {X<=9,X>=9};
  { X = 9, Y = 9 }
>    particular_value(X,   Y),{X<9};
  { X = 9-S$1,
    S$1 > 0,
    Y = 9 }
> particular_value(X,   X),{X<9};
>
```

Remarques :

Deux cas peuvent se présenter :

- Soit la valeur de *R1* qui satisfait le système de contraintes courant est unique. *R2* représente alors cette valeur.
- Soit l'ensemble des valeurs possibles qui satisfont le système de contraintes courant est infini. La valeur particulière choisie est alors indépendante de la contrainte ≠, et des contraintes d'inégalités strictes. On est cependant assuré qu'une valeur très proche de celle exhibée existe.

pgcd(N1, N2, N3)

Fonction	: Calcul du plus grand commun diviseur
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1, N2*
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N3'},* où *N3'* est le plus grand commun diviseur des entiers *N1* et *N2*. Le plus grand commun diviseur calculé ne tient pas compte du signe. *pgcd* échoue si *N1* et *N2* ne sont pas connus ou pas entiers.

Exemples :

```
> pgcd(24,16,P);
  { P = 8 }
> pgcd(24,-16,P);
  { P = 8 }
>
```

Voir également :

- *gcd*
- *ppcm, lcm*

© PrologIA

ppcm(N1, N2, N3)

Fonction	: Calcul du plus petit commun multiple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *N1, N2*
Messages d'erreur	: Non

Description :

Pose la contrainte *{N3 = N3'},* où *N3'* est le plus petit commun multiple des entiers *N1* et *N2*. *ppcm* échoue si *N1* et *N2* ne sont pas connus ou pas entiers.

Exemples :

```
> ppcm(12,18,P);
  { P = 36 }
> ppcm(12,-18,P);
  { P = -36 }
>
```

Voir également :

- *lcm*
- *pgcd, gcd*

predefined(T)

Fonction	: Teste si le terme T est une règle prédéfinie ou un prédicat évaluable
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: *T*
Messages d'erreur	: Non

Description :

S'efface avec succès si *T* est un terme correspondant à l'appel d'une règle prédéfinie ou un prédicat évaluable.

Exemples :

```
> predefined(list);
  {}
> predefined(predefined[U]);
  { U !tuple, U :: 1 }
> predefined(myrule);
>
```


© PrologIA

prologIII

Fonction	: Bascule en syntaxe Prolog III
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Bascule en syntaxe Prolog III si l'on se trouve en syntaxe Edimbourg. Se comporte en fait exactement de la même manière que l'appel de la primitive *set_config("syntax","Prolog III")* .

Voir également :

- *edinburgh, set_config, set_configuration*

quit

Fonction	: Quitter Prolog III
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Fait quitter Prolog III sans rien sauver.

Voir également :

- *quit(N)*

© PrologIA

quit(N)

Fonction	: Quitter Prolog III
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *N*
Messages d'erreur	: Non

Description :

Fait quitter Prolog III sans rien sauver. Si *N* a pour valeur 0, cette valeur est convertie dans le code correspondant à «pas d'erreur» pour le système d'exploitation; les autres valeurs de *N* ne sont pas converties. Ce code est ensuite envoyé au système d'exploitation à titre de “statut de terminaison”.

Remarques :

La forme, *quit*, équivaut à *quit(0)*.

Voir également :

- *quit*

rational(R)

Fonction	: Vérifie que R est un nombre rationnel connu
Classe	: Procédure externe Prolog III
Catégorie	: Vérifications de types
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

S'exécute avec succès si *R* représente une valeur rationnelle non entière et connue. Echoue dans tous les autres cas.

Exemples :

```
> rational(3004/1960);
  {}
> rational(31.897);
>
  rational(6/3);
>
  rational(X);
>
```


© PrologIA

real(F)

Fonction	: Vérifie que R est un nombre flottant connu
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Vérifications de types
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

S'exécute avec succès si *F* représente un nombre flottant connu. Echoue dans tous les autres cas.

Exemples :

```
> real(3004.1960e20);
  {}
>
  real(31/897);
>
  real(1.0);
  {}
> real(X+2.0);
>
```

redef_array(I,N)

Fonction	: Modifie la taille d'un tableau
Classe	: Procédure externe Prolog III
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I, N*
Messages d'erreur	: Voir ci-après

Description :

Modifie la taille allouée à un tableau créé par la primitive *def_array*.

Exemples :

```
> def_array(tab,100)  assign(tab(50),3);
  {}
> val(tab(50),X);
  { X = 3 }
> redef_array(tab,200)   assign(tab(154),2);
  {}
> val(tab(154),X);
  { X = 2 }
> redef_array(tab,40);
  {}
> val(tab(50),X);
Erreur 246: Erreur dans val
>
```

Voir également :

- *assign, def_array, undef_array, val*

© PrologIA

reinsert

Fonction	: Insertion de règles à partir de l'unité d'entrée
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: Aucun
Messages d'erreur	: Voir ci-après

Description :

Cette règle prédéfinie fait basculer le système dans un mode pour lequel les règles lues sur l'unité d'entrée courante sont ajoutées au programme en cours, dans l'ordre dans lequel elles sont lues. Le mode *reinsert* se termine soit quand une règle vide est trouvée, soit si l'on rencontre la fin du fichier d'entrée.

Exemple :

```
> reinsert;
conc([],y,y)  ->;
conc([e|x],y,[e|z])  ->  conc(x,y,z);;
>
```

Si une erreur de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un « ; », ou jusqu'à la fin de la ligne sur l'unité *console*) sont ignorés. Contrairement à la primitive *insert*, *reinsert* remplace un paquet déjà existant par sa nouvelle définition. Attention ceci peut être dangereux : par exemple une erreur sur une règle à l'intérieur d'un paquet peut provoquer, avec *reinsert*, l'écrasement des règles précédentes quand on continuera à lire le paquet.

Voir également :

- *insert*

reinsert(F)

Fonction	: Insertion de règles à partir d'un fichier
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *F*
Messages d'erreur	: Voir ci-après

Description :

Insère des règles à partir du fichier de nom *F*. La seule différence avec le prédicat *insert* est que si un paquet existe déjà, *reinsert* remplace l'ancien paquet par sa nouvelle définition. Attention ceci peut être dangereux : une erreur sur le nombre d'argument d'une règle à l'intérieur d'un paquet, provoquera avec *reinsert* l'écrasement des règles précédentes quand on continuera à lire le paquet.

Voir également :

- *insert*

© PrologIA

reload(F, L)

Fonction	: Chargement de modules sauvés
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Entrées/sorties
Paramètres connus	: *F*, *L*
Messages d'erreur	: Chaîne attendue, Argument de mauvais type

Description :

Chargement de modules sauvés. *F* (chaîne de caractères) est un nom de fichier et *L* est une « liste de substitution de préfixes ». Cette commande produit le chargement des modules sauvegardés dans le fichier indiqué; celui-ci doit avoir été produit par une des commandes *save* ou *exit*.

L est une liste de la forme [<$pref_1$, $subs_1$> , … , <$pref_k$, $subs_k$>] qui spécifie le renommage des modules chargés : $pref_1$ sera substitué par $subs_1$, $pref_2$ par $subs_2$, etc…

Exemple :

```
> reload("myfile.mo",[<"data","donnees">]);
  {}
>
```

Remarques :

Une tentative de chargement d'un élément (règle ou tableaux) déjà connu donne lieu au remplacement de l'ancienne version par la nouvelle.
Les modules sauvegardés sont sous forme de *code objet* (et non texte).

remove_implicit(S1, S2)

Fonction	: Modification de contexte
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *S1, S2*
Messages d'erreur	: Voir ci-après

Description :

Enlève l'identificateur ayant le nom abrégé *S2* de la partie fermée de la famille correspondant au préfixe *S1*.

Voir la section « *Structuration, saisie et modification des règles* » du chapitre « *Le contrôle et l'environnement des programmes* » pour plus de détails.

Voir également :

- *add_implicit, dictionary*
- *set_context, current_context, close_context_dictionary*
- *module, end_module*

© PrologIA

repeat

Fonction	: Boucles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Il s'agit simplement du prédicat suivant :

```
repeat -> ;
repeat -> repeat;
```

reset_cpu_time

Fonction	: Initialisation du chronomètre
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Réinitialise le chronomètre.

Voir également :

- *cpu_time*

© PrologIA

resume_trace

Fonction	: Relance la trace si celle-ci a été inhibée par la primitive *suspend_trace*
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Relance la trace, si celle-ci, active, a été inhibée par la primitive *suspend_trace*. La réunion de ces deux primitives est extrêmement utile lorsque l'on désire supprimer la trace de certains prédicats particuliers. Il suffit alors d'encadrer ces prédicats par les primitives *suspend_trace* et *resume_trace*. Si aucune trace n'est active, les appels à *suspend_trace* et *resume_trace* s'exécutent sans actions particulières.

Exemples :

```
> essai  ->
     suspend_trace
     outml("sans  trace")
     resume_trace  ;
> trace  line  essai  line;
line

essai = essai
suspend_trace
sans trace
line

   {}
> no_trace;
no_trace
{}
> essai;
sans trace
   {}
>
```

Voir également :

- *trace, no_trace, set_config, suspend_trace*

retract(T,Q)

Fonction	: Suppression des règles correspondant à un modèle donné.
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Recherche et supprime les règles dont la tête s'unifie avec *T* et dont les buts s'unifient avec les éléments de la liste *Q*. Le terme *T* doit cependant être suffisamment connu pour qu'il n'existe aucune ambiguïté quant au nom des règles visées.

Exemples :

```
> toto(aa)  ->;
> toto(bb)  ->;
> toto(x)  ->  titi(x,y)  tutu(z,y);
> retract(toto(X),[]);
  { X = aa }
  { X = bb }
> list;
toto(x) ->
    titi(x,y)
    tutu(z,y) ;
  {}
> retract(toto(aa),[titi[U],  tutu[V]]);
  { U = <aa,y_8>, V = <z_8,y_8> }
> list;
  {}
>
```

Voir également :

- *suppress, assert, list, rule*

© PrologIA

rule(T,Q)

Fonction	: Recherche de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Règle inexistante

Description :

Recherche des règles correspondant à un *modèle* donné. Cette primitive s'efface autant de fois qu'il existe de règles dont la tête s'unifie avec *T* et la queue avec *Q*. Si une telle règle n'existe pas, alors *rule(T,Q)* produit une erreur.

Exemple :

```
> list;
conc([],L,L)->;
conc([A/L],L',[A/L'']) ->
  conc(L,L',L'');
> rule(conc(X,Y,Z),Q);
  { X = [], Z = Y, Q = [] }
  { X = [v149/v150], Z = [v149/v151],
    Q = [conc(v150,y,v151)] }
> rule(concatener(X,Y,Z),Q);
>
```

Remarque :

T doit être de type **!idt**.

rule(N,T,Q)

Fonction	: Recherche de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Règle inexistante

Description :

Recherche des règles correspondant à un *modèle* donné. Cette primitive recherche la règle dont la tête s'unifie avec *T*, la queue avec *Q* et qui est de rang *N* dans son paquet. Si une telle règle n'existe pas, alors *rule(N,T,Q)* produit une erreur.

Exemple :

```
> list;
conc([],L,L)->;
conc([A/L],L',[A/L'']) ->
  conc(L,L',L'');
> rule(2,conc(X,Y,Z),Q);
  { X = [v149/v150], Z = [v149/v151],
    Q = [conc(v150,y,v151)] }
>
```

Remarque :

T doit être de type **!idt**.

© PrologIA

rule(N,A,T,Q)

Fonction	: Recherche de règles
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *A*
Messages d'erreur	: Règle inexistante

Description :

Recherche des règles correspondant à un *modèle* donné. Cette primitive s'efface autant de fois qu'il existe de règles dont la tête ou encore l'identificateur de tête s'unifie avec *A*. On peut affiner le modèle en précisant la tête *T*, la queue *Q* et le rang dans le paquet *N*. Si une telle règle n'existe pas, alors *rule(N,A,T,Q)* produit une erreur.

Exemple :

```
> list;
conc([],L,L)->;
conc([A/L],L',[A/L'']) ->
  conc(L,L',L'');
> rule(1,conc,T,Q);
  { T = conc([],v149,v149), Q = [] }
> rule(2,conc(X,Y,Z),T,Q);
  { X = [v149/v150], Z = [v149/v151],
     T = conc([v149/v150],y,[v149/v151])
     Q = [conc(v150,y,v151)] }
>
```

Remarque :

A doit être de type **!idt**.

rule_nb(<I,A>,N)

Fonction	: Nombre de règles d'un paquet
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I,A*
Messages d'erreur	: Règle inexistante

Description :

Compte les règles du paquet déterminé par l'identificateur *I* et l'arité *A*. *N* est alors unifié avec le nombre de règles de ce paquet. Echoue si *I* et *A* ne sont pas connus ou si le paquet n'existe pas.

Exemple :

```
> list;
conc([],L,L)->;
conc([A/L],L',[A/L'']) ->
  conc(L,L',L'');
> rule_nb(<conc,3>,N);
  { N = 2 }
>
```

Voir également :

- *rule, current_predicate*

© PrologIA

save

Fonction	: Sauvegarde de modules
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Sauvegarde de modules. Cette commande produit la sauvegarde dans le fichier de nom **prolog3.psv** de tous les éléments (règles, variables statiques et tableaux) de tous les modules.

Remarque :

Le fichier produit est un fichier de *code objet*, destiné à être ultérieurement chargé dans un autre environnement Prolog III au moyen de la commande *reload* .

Voir également :

- *reload*
- *exit, quit*

save(S)

Fonction	: Sauvegarde de modules
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *S*
Messages d'erreur	: Chaîne attendue

Description :

Sauvegarde de modules. Cette commande produit la sauvegarde dans le fichier de nom *S* (chaîne de caractères) de tous les éléments (règles, variables statiques et tableaux) de tous les modules.

Remarque :

Le fichier produit est un fichier de *code objet*, destiné à être ultérieurement chargé dans un autre environnement Prolog au moyen de la commande *reload* .

Voir également :

- *reload*
- *exit, quit*

© PrologIA

save(L,S)

Fonction	: Sauvegarde de modules
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *L,S*
Messages d'erreur	: Chaîne attendue, Fin de liste attendue, Préfixe inexistant

Description :

Sauvegarde de modules. *S* est un nom de fichier (chaîne de caractères) et *L* une liste de préfixes (chaînes de caractères). Cette commande produit la sauvegarde dans le fichier indiqué de tous les éléments (règles, variables statiques et tableaux) des modules correspondant aux préfixes donnés dans la liste *L*.

Exemple:

```
> save(["","data","dict"],  "myfile.mo");
  {}
>
```

Remarque :

Le fichier produit est un fichier de *code objet*, destiné à être ultérieurement chargé dans un autre environnement Prolog III au moyen de la commande *reload* . Ce n'est donc pas un fichier texte et il ne faut utiliser ni *insert* ni *input* avec ce fichier.

Voir également :

- *reload*
- *exit, quit*

set_config(S, V)

Fonction	: Modification de certains paramètres
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *S, V*
Messages d'erreur	: Non

Description :

Cette primitive permet de spécifier le comportement de Prolog III pour les points indiqués ci-dessous. En regard de chaque point, on indique la chaîne à fournir comme premier paramètre :

- le mode de fonctionnement de l'interpréteur "**machine_mode**"
- l'impression des termes sous `debugger` "**tracemode**"
- l'écho "**echo**"
- l'exécution d'un prédicat non défini "**undefined_rule**"
- la gestion de l'interruption de programme "**interrupt_is_error**"
- la syntaxe utilisée "**syntax**"
- la taille maximale d'une ligne pour l'affichage "**tty_wrap**"
- le format numérique pour l'affichage "**format_out_num**"
- le format d'affichage des flottants "**format_float**"
- le nombre de chiffres après le point décimal "**format_decimal**"
- l'erreur relative dans les calculs sur les flottants "**epsilon_float**"
- les statistiques liées à l'exécution "**statistics**"

Voici à présent les valeurs possibles pour chacun de ces paramètres, ainsi que les valeurs par défaut :

machine_mode :

- 0 : Fonctionnement normal (sans `debugger` ni `trace`).
- 1 : Fonctionnement en mode `trace`.
- 2, 3 : Fonctionnement en mode `debugger`.

Valeur par défaut : 0

© PrologIA

tracemode :

- 1 : Trace de niveau 1 : affichage du nom du prédicat.
- 2 : Trace de niveau 2 : affichage du nom du prédicat et de ses arguments.
- 3 : Trace de niveau 3 : affichage du nom du prédicat, de ses arguments et du système de contraintes relatif aux variables qui interviennent dans les arguments.

Valeur par défaut : 2

echo :

- 0 : Pas d'écho.
- 1 : Affichage sur l'unité courante de sortie de toutes les règles ou requêtes lues sur l'unité courante d'entrée. Ces règles, lorsqu'elles sont valides, sont d'abord codées sous une forme canonique, puis ensuite décodées et listées sur la sortie courante.

Valeur par défaut : 0

undefined_rule :

Détermine le comportement de l'interpréteur lors de l'appel d'un but non exécutable :

- "fail" : Echec Prolog sans message d'avertissement ni erreur.
- "warning" : Echec Prolog, avec message d'avertissement (paquet non défini ou but non exécutable).
- "error" : Produit une erreur (`Accès non défini`).

Valeur par défaut : "warning"

interrupt_is_error :

- 0 : L'interruption de programme (<Ctrl-C> sur certaines machines) n'est pas traitée comme une erreur Prolog. Un dialogue, décrit dans le manuel d'utilisation, est présenté à l'utilisateur, et lui permet de continuer, de stopper, de tracer l'exécution du programme ou de sortir de la session Prolog.
- 1 : L'interruption de programme génère une erreur Prolog, avec le message d'erreur : “Err 16: Interruption utilisateur”. Dans ce cas l'interruption peut être interceptée par le mécanisme de *block*.

Valeur par défaut : 0

syntax :

- "Prolog III" : Mode syntaxique de type Prolog III (II+).
- "Edinburgh" : Mode syntaxique de type Edimbourg.

Valeur par défaut : "Prolog III"

Remarque :

Il est possible de se retrouver directement sous le mode syntaxique Edimbourg, pour chaque session Prolog III. Pour cela, il suffit une première fois de positionner l'option "**syntax**" à "Edinburgh", puis de sauver l'état courant à l'aide de la commande *exit*. On invitera chaque utilisateur à renommer auparavant le fichier **initial3.psv** afin d'en conserver une version à l'état d'origine.

tty_wrap :

- Un entier compris entre 10 et 4000 représentant la longueur maximale admise pour l'écriture d'une ligne sur l'unité de sortie courante.

Valeur par défaut : 130

format_float :

- Une chaîne représentant un format d'entrée/sortie accepté par la fonction *printf()* du langage C. Cette chaîne doit de plus commencer par le caractère % et contenir moins de 19 caractères.

Valeur par défaut : "%.15le"

format_out_num :

- "Fnormal" : Affichage des rationnels en précision infinie, et des flottants en notation scientifique.
- "Fexact" : Affichage des rationnels en précision infinie, les flottants sont convertis en rationnels.
- "Fdecimal" : Affichage sous forme décimale.
- "Ffloat" : Affichage en notation scientifique.
- "Fint+fract" : Affichage sous la forme partie entière + partie fractionnaire si cette dernière est non nulle.

Valeur par défaut : "Fnormal"

Exemples :

```
> set_config("format_out_num","Fnormal")
    {X = 120/36, Y = 1.5};
```


© PrologIA

```
  { X = 10/3, Y = 1.500000000000000e+00 }
> set_config("format_out_num","Fexact")
    {X = 120/36, Y = 1.5};
  { X = 10/3, Y = 3/2 }
> set_config("format_out_num","Fdecimal")
    {X = 120/36, Y = 1.5};
  {X = 3.333, Y = 1.5 }
> set_config("format_out_num","Ffloat")
    {X = 120/36, Y = 1.5};
  {X = 3.333333333333333e+00, Y = 1.500000000000000e+00 }
> set_config("format_out_num","Fint+fract")
    {X = 120/36, Y = 1.5};
  { X = (3+1/3), Y = (1+1/2) }
>
```

format_decimal :

- Un entier compris entre 0 et 7 représentant le nombre de chiffres affichés après le point (partie décimale). Cette option est utilisée quand l'option "**format_out_num**" est positionnée à "Fdecimal" .

Valeur par défaut : 3

epsilon_float :

- Nombre flottant compris entre 10^{-4} et 10^{-12} représentant l'intervalle de tolérance relatif, admis lors de la comparaison de nombres flottants.

Valeur par défaut : 10^{-7}

statistics :

- 0 : Pas de statistiques.
- 1 : Affiche les statistiques liées à l'exécution.

Valeur par défaut : 0

Voir également :

- *set_configuration, get_config, get_configuration*
- *echo, no_echo, trace, no_trace, debug, no_debug*
- *edinburgh, prologIII*

set_configuration(U)

Fonction	: Modification de certains paramètres
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *U*
Messages d'erreur	: Non

Description :

Même fonctionnalité que *set_config*, mais *U* est un tuple de doublets *<champ, valeur>*. On peut donc spécifier les valeurs de plusieurs champs simultanément.

Exemple :

```
> set_configuration(<<"syntax","Edinburgh">,
                           <"format_out_num","Ffloat">>);
  {}
?- {X = 12}.
  { X = 1.200000000000000e+01 }
?-
```

Voir également :

- *set_config, get_config, get_configuration*

© PrologIA

set_context (I, S, S', D)

Fonction	: Description et activation d'un contexte
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I, S, S', D*
Messages d'erreur	: Non

Description :

Cette commande permet à la fois de décrire un contexte, et de l'installer comme le contexte courant de lecture/écriture. Lorsqu'un nouveau contexte est activé, l'ancien contexte n'est plus pris en compte.

Voir la section « *Structuration, saisie et modification des règles* » du chapitre « *Le contrôle et l'environnement des programmes* » pour la signification des arguments :

I : nom du contexte nouvellement défini
S : suite explicite
S' : suite implicite
D : préfixe par défaut

Lorsque Prolog III est lancé, le contexte courant de lecture/écriture est le contexte défini par la commande :

```
set_context("", [], ["sys"], "")
```

set_ident (I, N)

Fonction	: Affectation d'un attribut à un identificateur
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I, N*
Messages d'erreur	: Oui, voir le manuel d'utilisation

Description :

Cette primitive permet d'associer un nombre et un identificateur. Elle est utilisée principalement pour associer un nom logique et un nombre utilisé dans le fichier *userrule.c* pour définir une procédure externe. On consultera le manuel d'utilisation pour plus de détails concernant la définition de primitives par l'utilisateur.

© PrologIA

size(U, N)

Fonction	: Taille d'un tuple
Classe	: Règle prédéfinie Prolog III
Catégorie	: Arbres, listes, chaînes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Pose la contrainte *{U :: N}*, avec les retardements nécessaires si *N* ou *U* ne sont connus.

Voir également :

- *bound_size*

split(U, L)

Fonction	: Décomposition d'un tuple
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversion
Paramètres connus	: *U*
Messages d'erreur	: Non

Description :

Décompose un tuple (ou, donc, une chaîne) en la liste de ses éléments, c'est à dire pose la contrainte *{L = U'}*, où *U'* est la liste des éléments du tuple *U*. Echoue si *U* n'est pas connu à l'exécution.

Exemples :

```
> split("abcde",V);
  { V = [`a`,`b`,`c`,`d`,`e`] }
> split(<1,2,3>,V);
  { V = [1,2,3] }
> split(U,V),  {U  !tuple,  U::3};
  { U = <X1,X2,X3>, V = [X1,X2,X3] }
>
```

Nous avons renommé les variables intermédiaires créées par Prolog III.

Voir également :

- *list_string, list_tuple*

© PrologIA

spy(<I, N>)
spy(S)

Fonction	: Espionnage de paquets de règles
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *S*, au moins un parmi *I* et *N*
Messages d'erreur	: Module inexistant

Description :

Surveille un ou plusieurs paquets de règles en installant des points-espion .

Dans la première forme, *I* désigne l'identificateur de paquet et *N* l'arité de ce paquet, et au moins l'une des deux informations doit être connue à l'appel. Un échec survient dans le cas contraire.

Dans la deuxième forme, *S* est un nom de module sous la forme d'une chaîne de caractère. Toutes les règles du module *S* sont alors concernées. Provoque une erreur s'il n'existe aucune règle de préfixe *S*.

L'utilisateur se réfèrera au manuel “Mise au point de programmes Prolog III” pour obtenir la définition et l'emploi des points-espion.

Voir également :

- *no_spy*
- *debug, no_debug*

© PrologIA

state

Fonction	: Affiche l'occupation de certaines zones Prolog
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Affiche l'état courant des piles.

Exemple :

```
> state;
copy stack           : 977/799900 lwords        0 %
back stack           : 0/200000 lwords          0 %
rule space           : 48235/100000 lwords      48 %
dictionary (acces)   : 20556/36864 bytes        55 %
dictionary (names)   : 4053+12144/32784 bytes   49 %
  {}
>
```

string(S)

Fonction	: Vérifie que *S* est une chaîne connue
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Vérifications de type
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

- S'exécute avec succès si *S* représente une chaine entièrement connue. Echoue dans tous les autres cas.

Exemples :

```
> string("a  l'abordage");
  {}
> string(<`C`,`I`,`A`>);
  {}
> string("debut".<A,B,C>);
>
  string(`X`);
>
  string(not_a_string);
>
```


© PrologIA

string_bool(S, B)

Fonction	: Conversion d'une chaîne en un booléen
Classe	: Règle prédéfinie Prolog III
Catégorie	: Conversions
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

- Si *S* est connu : pose l'ensemble de contraintes *{B = S'}* où *S'* est un booléen formé à partir des caractères de la chaîne *S*.

- Si *B* est connu : pose l'ensemble de contraintes *{S = B'}*, où *B'* est une chaîne formée des caractères qui composent le booléen *B*.

Exemples :

```
> string_bool(S,1');
  { S = "1'" }
> string_bool("0'",B);
  { B = 0' }
>
```

string_ident(S,I)

Fonction	: Conversion d'une chaîne en un identificateur
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

- Si *S* est connu : pose l'ensemble de contraintes *{I = I'}*, où *I'* est un identificateur composé des caractères de la chaîne qui le représente.

- Si *I* est connu : pose l'ensemble de contraintes *{S = S'}* où *S'* est une chaîne composée des caractères de l'identificateur *I*.

Exemples :

```
> string_ident(S,histoire:valmy);
  { S = "histoire:valmy" }
> string_ident("histoire:valmy",I);
  { I = histoire:valmy }
>
```


© PrologIA

string_ident(P,S,I)

Fonction	: Conversion d'une chaîne en un identificateur
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

- Si *I* est connu : pose l'ensemble de contraintes *{S = S', P = P'}* où *S'* est une chaîne composée des caractères de la représentation abrégée de l'identificateur *I,* et *P'* une chaîne composée des caractères formant le préfixe de l'identificateur *I.*

- Si *S* et *P* sont connus : pose l'ensemble de contraintes *{I = I'},* où *I'* est un identificateur composé des caractères de la chaîne représentant son préfixe, *P,* et de celle représentant sa notation abrégée, *S*.

Exemples :

```
> string_ident(P,S,histoire:valmy);
  { P = "histoire", S = "valmy" }
> string_ident("histoire","valmy",I);
  { I = histoire:valmy }
> string_ident("geographie:fleuve","Seine",I);
  { I = 'geographie:fleuve:Seine' }
> string_ident("geographie","fleuve:Seine",I);
>
```

string_integer(S,N)

Fonction	: Conversion d'une chaîne en un entier
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: Voir ci-après
Messages d'erreur	: Entier attendu

Description :

- Si *S* est connu : pose l'ensemble de contraintes *{I = S'}* où *S'* est un entier formé à partir des caractères de la chaîne *S*.

- Si *N* est connu : pose l'ensemble de contraintes *{S = N'},* où *N'* est une chaîne formée des caractères qui composent l'entier *N*.

Exemples :

```
> string_integer("12",N);
  { N = 12 }
> string_integer(S,24/2);
  { S = "12" }
> string_integer("12.3",N);
Erreur 234 : entier attendu
>
```


© PrologIA

string_real(S,F)

Fonction	: Conversion d'une chaîne en un flottant
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Conversions
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

- Si *S* est connu : pose l'ensemble de contraintes *{F = S'}* où *S'* est un flottant formé à partir des caractères de la chaîne *S.*

- Si *F* est connu : pose l'ensemble de contraintes *{S = F'},* où *F'* est une chaîne formée des caractères qui composent le flottant *F.*

Exemples :

```
> string_real("3.14e5",F);
  { F = 3.140000 e+05 }
> string_real(S,3.14e5);
  { S = "3.140000 e+05" }
>
```

substring(S1, N1, N2, S2)

Fonction	: Extraction de sous-chaînes
Classe	: Règle prédéfinie Prolog III (II+)
Catégorie	: Arbres, listes chaînes et tuples
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Pose la contrainte *{S2 = S1'},* où *S1'* est la sous-chaîne de *S1* commençant à la position *N1,* de longueur *N2*. Echoue si une telle sous-chaîne n'existe pas ou si *S1* n'est pas connue.

Exemple :

```
> substring("1234567890",3,4,X);
  { X = "3456" }
>
```


© PrologIA

suppress(I)

Fonction	: Suppression d'un paquet de règles
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I*
Messages d'erreur	: Aucun

Description :

Supprime tout le paquet de règles de nom *I*. Echoue si *I* est inconnu, ou bien s'il n'existe pas de paquet de règles convenant.

Exemples :

```
> essayons(1)->;
> essayons(2)->;
> essayons(1,2)->;
> essayons(X);
  { X = 1 }
  { X = 2 }
> suppress(essayons);
  {}
> essayons(X);
>
  essayons(X,Y);
>
```

suppress(I, A)

Fonction	: Suppression d'un paquet de règles
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I* et *A*
Messages d'erreur	: Aucun

Description :

Supprime tout le paquet de règles de nom *I*, et d'arité *A*. Echoue si *I* ou *A* sont inconnus, ou bien s'il n'existe pas de paquet de règles convenant.

Exemples :

```
> essayons(1)->;
> essayons(2)->;
> essayons(1,2)->;
> essayons(X);
  { X = 1 }
  { X = 2 }
> suppress(essayons,1);
  {}
> essayons(X);
>
  essayons(X,Y);
  { X = 1,Y = 2 }
>
```


© PrologIA

suppress(I, A, N)

Fonction	: Suppression d'une règle
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I*, *A*, et *N*
Messages d'erreur	: Aucun

Description :

Supprime la règle numéro *N* du paquet de règles de nom *I*, et d'arité *A*. Réussit, mais sans rien effacer, si aucune règle ne correspond. Echoue si les arguments sont inconnus ou de mauvais type.

Exemples :

```
> essayons(1)->;
> essayons(2)->;
> essayons(1,2)->;
> essayons(X)  essayons(Y,Z);
  { X = 1, Y = 1, Z = 2 }
  { X = 2, Y = 1, Z = 2 }
> suppress(essayons,1,2);
  {}
> essayons(X)  essayons(Y,Z);
  { X = 1, Y = 1, Z = 2 }
>
```

suspend_trace

Fonction	: Annule la trace si celle-ci est active
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Annule la trace, si celle-ci est active jusqu'au prochain *resume_trace*. La réunion de ces deux primitives est extrêmement utile lorsque l'on désire supprimer la trace de certains prédicats particuliers. Il suffit alors d'encadrer ces prédicats par les primitives *suspend_trace* et *resume_trace*. Si aucune trace n'est active, les appels à *suspend_trace* et *resume_trace* n'ont pas d'effets particuliers.

Exemples :

```
> essai  ->
     suspend_trace
     outml("sans  trace")
     resume_trace  ;
>    trace  line  essai  line;
line

essai = essai
suspend_trace
sans trace
line

   {}
> no_trace;
no_trace
   {}
> essai;
sans trace
   {}
>
```

Voir également :

- *trace, no_trace, set_config, resume_trace*

© PrologIA

sys_command(S)

Fonction	: Exécution d'une commande du système d'exploitation
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: *S*
Messages d'erreur	: Aucun

Description :

Exécute une commande du système d'exploitation, représentée par la chaîne *S*. Consultez le manuel d'utilisation relatif à votre système d'exploitation pour connaître la liste et la syntaxe des commandes disponibles.

trace

Fonction	: Trace de l'exécution d'un programme
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Environnement
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Mode de trace d'un programme, avec impression, sur l'unité courante de sortie de trace, des contraintes d'égalité entre les buts et les têtes de règles avant unification, et visualisation des points de backtracking.

Cette primitive *trace* correspond à l'appel :

```
set_config("machine_mode", 1)
```

La primitive *no_trace* permet de sortir de ce mode.

Voir également :

- *no_trace, debug, no_debug, trace_file, suspend_trace, resume_trace*
- *set_config, set_configuration, get_config, get_configuration,*

© PrologIA

trace_file(S)

Fonction	: Redirection de la trace
Classe	: Procédure externe Prolog III
Catégorie	: Environnement
Paramètres connus	: *S*
Messages d'erreur	: Non

Description :

Redirection, dans le fichier de nom déterminé par la chaîne de caractères *S*, de la trace d'exécution d'un programme.

Si la chaine de caractère est la chaine vide, la trace est redirigée sur l'unité de trace définie au démarrage de Prolog III.

Le fichier en question ne sera fermé qu'au prochain *trace_file* (on pourra par exemple utiliser *trace_file("")*)

Voir également :

- *trace, no_trace*

trunc(R, N)

Fonction	: Partie entière
Classe	: Procédure externe Prolog III
Catégorie	: Contraintes numériques
Paramètres connus	: *R*
Messages d'erreur	: Aucun

Description :

Pose la contrainte *{N = R'}*, où *R'* est la partie entière du nombre *R*.

Exemples :

```
> trunc(5/4,N);
  { N = 1 }
> trunc(-3.56,N);
  { N = -4 }
>
```


© PrologIA

tuple(U)

Fonction	: Vérifie que *U* est un tuple connu
Classe	: Procédure externe Prolog III
Catégorie	: Vérifications de types
Paramètres connus	: *U*
Messages d'erreur	: Aucun

Description :

S'exécute avec succès si *U* représente un tuple connu, c'est-à-dire dont on sait si le nombre de fils est nul ou pas. Echoue dans tous les autres cas.

Exemples :

```
> tuple(<>);
  {}
> tuple(<1,0',ceslawi>);
  {}
> tuple("connu");
  {}
> tuple(<>[X]);
>
  tuple(<A,B,C>);
  {}
> tuple(U),  {U  !tuple};
>
```

undef_array(I)

Fonction	: Désalloue un tableau créé par *def_array*
Classe	: Procédure externe Prolog III
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I*
Messages d'erreur	: Voir ci-après

Description :

Désalloue un tableau créé par la primitive *def_array* et récupère la place correspondante.

Exemples :

```
> def_array(tab,100)  assign(tab(50),3);
  {}
> val(tab(50),X);
  { X = 3 }
> undef_array(tab);
  {}
> val(tab(50),X);
Erreur 246: Erreur dans val
>
  undef_array(X11);
Erreur 233: Identificateur attendu (2)
>
```

Voir également :

- *assign, def_array, redef_array, val*

© PrologIA

val(T1, T2)

Fonction	: Evalue une expression construite à partir de fonctions évaluables
Classe	: Procédure externe Prolog III (II+)
Catégorie	: Contrôle
Paramètres connus	: Voir ci-après
Messages d'erreur	: Voir ci-après

Description :

Evalue l'expression *T1* et produit le résultat *T2*. L'expression à évaluer est construite récursivement à partir des constantes, des identificateurs, des éléments de tableau et des fonctions évaluables.

Exemples :

```
> val(add(mul(2,add(3,4)),1000),x);
  { X = 1014 }
> val(2*(3+4)+1000,x);
  { X = 1014 }
>
```

Remarques :

Chaque fonction évaluable doit avoir des arguments d'un type précis : il n'y a pas de conversion automatique de type. Quand une fonction a un nombre incorrect d'arguments ou quand certains de ses arguments sont du mauvais type, *val* produit une erreur.

• La valeur d'un nombre ou d'une chaîne est égale à ce nombre ou à cette chaîne.

• La valeur d'un tableau (externe ou interne) indicé est égale à la valeur de l'élément correspondant de ce tableau.

Exemples:

```
> def_array(tab,100)  assign(tab[50],3);
  {}
> val(tab[50],X);
  { X = 3 }
>
```

• La valeur d'un identificateur *i* est définie comme suit :

(1) si une constante *k* a été affectée à *i* (au moyen de la règle *assign*) alors la valeur de *i* est *k*.
(2) si *i* n'a pas fait l'objet d'une affectation préalable, alors la valeur de *i* est *i* lui-même.

Exemples:

```
> assign(un,  1);
  {}
> val(un,  x)  val(deux,  y);
  { x=1,y=deux }
>
```

Voici toutes les fonctions évaluables :

add(t_1, t_2)
valeur(add(t_1, t_2)) = valeur(t_1) + valeur(t_2).
Les valeurs de t_1 et t_2 doivent être numériques.

sub(t_1, t_2)
valeur(sub(t_1, t_2)) = valeur(t_1) - valeur(t_2).
Les valeurs de t_1 et t_2 doivent être numériques.

mul(t_1, t_2)
valeur(mul(t_1, t_2)) = valeur(t_1) × valeur(t_2).
Les valeurs de t_1 et t_2 doivent être numériques.

div(t_1, t_2)
$valeur(div(t_1, t_2)) = \frac{valeur(t_1)}{valeur(t_2)}$ Il s'agit du quotient entier par défaut.
Les valeurs de t_1 et t_2 doivent être des nombres entiers.

© PrologIA

mod(t_1, t_2)
valeur(mod(t_1, t_2)) = valeur(t_1) modulo valeur(t_2).
Les valeurs de t_1 et t_2 doivent être des nombres entiers.

trunc(t_1)
valeur(trunc(t_1)) = partie entière de valeur(t_1).
La valeur de t_1 doit être numérique.

eql(t_1, t_2)
valeur(eql(t_1, t_2)) = si valeur(t_1) = valeur(t_2) alors 1 sinon 0.
Les valeurs de t_1 et t_2 doivent être des constantes de même type.

inf(t_1, t_2)
valeur(inf(t_1, t_2)) = si valeur(t_1) < valeur (t_2) alors 1 *sinon* 0.
Les valeurs de t_1 et t_2 doivent être des constantes de même type. Pour les entiers et les réels, on prend la relation "<" entre les nombres. Pour les chaînes, on prend l'ordre lexicographique et pour les identificateurs on prend l'ordre lexicographique sur les chaînes associées.

infe(t_1, t_2)
valeur(infe(t_1, t_2)) = si valeur(t_1) ≤ valeur (t_2) alors 1 *sinon* 0.
Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

sup(t_1, t_2)
valeur(sup(t_1, t_2)) = si valeur(t_1) > valeur (t_2) alors 1 *sinon* 0.
Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

supe(t_1, t_2)
valeur(supe(t_1, t_2)) = si valeur(t_1) ≥ valeur (t_2) alors 1 *sinon* 0.
Les valeurs de t_1 et t_2 doivent être des constantes de même type. Voir *inf*.

if(t, t_1, t_2)
valeur(if(t, t_1, t_2)) = si (valeur(t)≠0) alors valeur(t_1) sinon valeur(t_2)

abs(t)
valeur(abs(t)) = valeur absolue de valeur(t).

Les fonctions suivantes donnent un résultat de type réel. Les fonctions trigonométriques travaillent avec des angles exprimés en radians.

atan(t)
valeur (atan(t)) = arc tangente de valeur(t).

cos(t)
valeur (cos(t)) = cosinus(valeur(t)).

exp(t)
valeur (exp(t)) = exponentielle(valeur(t)).

ln(t)
valeur(ln(t)) = logarithme népérien(valeur(t)).

rad(t)
valeur (rad(t)) = conversion en radian(valeur(t)).

sin(t)
valeur(sin(t)) = sinus(valeur(t)).

sqrt(t)
valeur(sqrt(t)) = racine carrée(valeur(t)).

tan(t)
valeur(tan(t)) = tangente(valeur(t)).

Voir également :

- *assign, def_array, undef_array, redef_array*

© PrologIA

Index des règles prédéfinies, procédures externes et fonctions évaluables

abs 369
add 368
add_implicit 202
arg 203
arg2 204
arg3 205
assert 206
assert'' 208
asserta 207
assertz 208
assign 209
atan 370
block 212
block_exit 212
bool 214
bound 215
bound_conc 216
bound_mult 217
bound_size 218
bound_tree 219
bound_tree' 220
char 221
char_code 222
close_context_dictionary 223
close_input 224
close_output 225
conc3 226
conc_string 227
cos 370
cpu_time 228
current_context 229
current_predicate 230
cut 231
def_array 232
default 234
dictionary 235
dif 237
div 238; 368
dot 239
echo 240; 341
edinburgh 241
edit 242
end_module 243
enum 244
epsilon_float 342
eq 246
eql 369
exit 247
exp 370
fail 249
find_pattern 251
findall 250
format_decimal 342
format_float 342
format_out_num 341
Fdecimal 341
Fexact 341
Ffloat 341
Fint+fract 341
Fnormal 341
free 252
free_label 253
freeze 254
garbage_collection 255
gcd 256
get_config 257

get_configuration 258
getenv 259
ident 260
if 369
in_char 261
in_char' 262
in_ident 263
in_integer 264
in_real 265
in_sentence 266
in_sentence' 267
in_string 268
in_term 269
inf 369
infe 369
inl 270
input 271
input_is 272
insert 273
integer 275
is_bool 276
is_char 277
is_ident 278
is_leaf 279
is_num 280
is_tuple 281
is_univ 282
kill_module 283
known 284
known_part 285
lcm 286
line 287
list 288
list_string 293
list_tuple 294
ln 370
max_value 295
min_value 296
mod 297; 369
module 298
mul 368
mult 299
new 300
next_char 301
next_char' 302
no_echo 303
no_trace 304
not 305
num 306
numden 307
out 308
outc 309
outl 310
outm 311
outml 312
output 313
output_is 314
particular_value 315
pgcd 316
ppcm 317
predefined 318
prologIII 319
quit 320
rad 370
rational 322
real 323
redef_array 324
reinsert 325
reload 327
remove_implicit 328
repeat 329
reset_cpu_time 330
resume_trace 331
retract 332
rule 333
rule_nb 336

© PrologIA

save 337
set_config 340
  echo 341
  epsilon_float 342
  format_decimal 342
  format_float 342
  format_out_num 341
  statistics 342
  syntax 342
  trace 340
  tty_wrap 342
  undefined_rule 343
set_configuration 344
set_context 345
set_ident 346
sin 370
size 347
split 348
sqrt 370
state 349
string 350
string_bool 351
string_ident 352
string_integer 354
string_real 355
sub 368
substring 356
sup 369
supe 369
suppress 357
suspend_trace 360
syntax 342
  Edinburgh 342
  Prolog III 342
sys_command 361
tan 370
trace 340; 362
trace_file 363
trunc 364; 369
tty_wrap 342
tuple 365
undef_array 366
undefined_rule 343
  error 343
  fail 343
  warning 343
val 367
  abs 369
  add 368
  atan 370
  cos 370
  div 368
  eql 369
  exp 370
  if 369
  inf 369
  infe 369
  ln 370
  mod 369
  mul 368
  rad 370
  sin 370
  sqrt 370
  sub 368
  sup 369
  supe 369
  tan 370
  trunc 369

Association Prolog
HERITAGE

Règles prédéfinies et procédures externes pour la syntaxe Edimbourg

Que trouve-t-on dans ce chapitre ?

Ce chapitre regroupe par ordre alphabétique toutes les règles prédéfinies et disponibles en Prolog III depuis le seul mode Edimbourg .
La plupart de ces prédicats ayant été adaptés aux fonctionnalités spécifiques de Prolog III, la compatibilité n'est pas totale. Certains sont des synonymes des primitives décrites dans le chapitre précédent, d'autres sont plus originaux.

© PrologIA

Identificateur - Arité

Il est coutume en syntaxe Edimbourg de trouver au niveau de la gestion des règles des termes de la forme `I/N`, `I` représentant un identificateur et `N` un entier désignant une arité. Pour des raisons évidentes, il est impossible en Prolog III de former de tels termes (il n'est pas possible de diviser un identificateur par un entier). Il nous faut donc une autre convention pour désigner ces termes. La notation de tuple a été préférée. On définira donc le couple ident/arité par `<ident,arité>` .

Règles sous forme de termes

On manipule parfois les règles en tant que termes, la plupart du temps au sein de primitives de gestion de règles. Dans ces cas là, il faut se souvenir que les règles Prolog III ont quatre formes possibles,

tete (1)

tete { *contraintes* } (2)

tete :- *but*, …, *but* (3)

tete :- *but*, …, *but* { *contraintes* } (4)

et que l'opérateur composé «{}» est l'opérateur principal dans les formes (2) et (4) .

© PrologIA

abolish(I)
abolish(<I, A>)

Fonction	: Suppression d'un paquet de règles
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: L'argument doit être un identificateur connu ou un tuple *<ident connu, Arité connue>*
Message d'erreur	: Non

Description :

Première forme : Supprime tout le paquet de règles de nom *I*. Se comporte comme *suppress*(*I*) .

Deuxième forme : Supprime tout le paquet de règles de nom *I* dont l'arité est *A*. Se comporte alors comme *suppress*(*I*, *A*) . Echoue lorsque les arguments ne sont pas connus.

Voir également :

- *retract*

assert(T)
assertz(T)

Fonction	: Ajout d'une règle à la fin de son paquet
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Si *T* est une règle incorrecte

Description :

Ajout d'une règle, à la fin de son paquet. *T* doit être un terme ayant l'une des formes décrites en introduction. La tête de la règle doit être un arbre étiqueté par un identificateur connu.

L'effacement de *assert(T)* ajoute la règle ou le fait *T au-dessous* du groupe de règles ayant le même «nom» que la tête de *T*. Par exemple, les deux commandes suivantes

```
?-  assert(conc([],Y,Y),[]).
{}
?- assert(  conc([E|X],Y,[E|Z]):-  conc(X,Y,Z)  ).
{}
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
conc([], Y, Y).
conc([E|X], Y, [E|Z]) :- conc(X, Y, Z).
```

Ajouter des règles comportant un système de contraintes peut être réalisé de deux façons. Une première façon pose des contraintes en dehors de l'appel :

© PrologIA

```
?- assert(hors_de(X,[]),[]).
{}
?- assert(  hors_de(X,[Y|Q]):-hors_de(X,Q)  )  {X # Y}.
{ X # Y }
?- list.
hors_de(X,[]) .
hors_de(X,[Y | Q]) :-
  hors_de(X, Q)
  { X # Y }.
{}
```

Une seconde méthode consiste à passer toute la règle en argument (puisque le tout est un terme). Dans ce cas, *assert* traduit si possible les termes placés entre accolades en véritables contraintes. Ces contraintes-là ne sont pas conservées après l'appel.

```
?- assert(  hors_de(X,  [Y|Q]:-  hors_de(X,Q)  {X#Y}  ).
{}
?- list.
hors_de(X,[Y | Q]) :-
  hors_de(X, Q)
  { X # Y }.
{}
```

Dans le cas où la règle en argument contient des buts séparés par une ou plusieurs virgules, il faut parenthéser celle-ci pour lever l'ambiguïté entre les deux types de virgules (virgule d'arguments et virgule de queue de règle).

```
?- assert((a  :-  b,  c)).
{}
```

asserta(T)

Fonction	: Ajout d'une règle en début de son paquet
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Si *T* est une règle incorrecte

Description :

Ajout d'une règle, au début de son paquet. *T* doit être un terme ayant l'une des formes décrites en introduction. La tête de la règle doit être un arbre étiqueté par un identificateur connu.

L'effacement de *assert(T)* ajoute la règle ou le fait *T au-dessus* du groupe de règles ayant le même «nom» que la tête de *T*. Par exemple, les deux commandes suivantes

```
?- asserta(  conc([E|X],Y,[E|Z]):-  conc(X,Y,Z)  ).
{}
?-  asserta(conc([],Y,Y),[]).
{}
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
conc([], Y, Y).
conc([E|X], Y, [E|Z]) :- conc(X, Y, Z).
```

Toutes les remarques concernant *assertz* sont valables ici.

© PrologIA

atom(T)

Fonction	: Vérifie que *T* est un identificateur connu
Classe	: Primitive Edimbourg
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que *T* est un identificateur connu. Echoue sinon. *atom(T)* se comporte comme le but *ident(T)* .

atomic(T)

Fonction	: Vérifie que *T* est un atome
Classe	: Primitive Edimbourg
Catégorie	: Vérification de types
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Vérifie que *T* est un atome connu; c'est à dire vérifie d'une part que *T* est connu, et que d'autre part *T* est un nombre, ou un identificateur, ou un booléen, ou un caractère, ou bien le tuple vide <> . Echoue dans tous les autres cas. Notons qu'une chaîne de caractère n'est pas un atome puisqu'on peut la découper en caractères.

Voir également :

- *bound, known*

© PrologIA

call(P)

Fonction	: Effacement d'un but
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: *P*
Messages d'erreur	: Non

Description :

Exécute le but *P*.
La primitive *Call* est définie par la règle :

```
call(X) :- X .
```

clause(T, Q)

Fonction	: Obtient la tête et la queue d'une règle
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Non

Description :

Unifie *T* et *Q* respectivement avec la tête et la queue de toutes les clauses dont l'accès est défini par *T*. Se comporte comme *rule* à la structure de la queue près. Celle-ci est représenté par l'identificateur *sys:true* lorsqu'elle est vide, par un terme lorsqu'elle ne comporte qu'un seul but, par une structure peignée, d'étiquette `','` sinon.

Remarques :

Il n'est pas possible de "récupérer" sous forme de termes l'ensemble des contraintes de cette règle. En effet, ces contraintes sont posées dès l'unification de la tête, et portent donc sur les variables de *T* et de *Q* .

Exemples :

```
?- consult("user").
differents(A, B) { A#B }.
regle(A,B) :- a(A) { A#B }.
longue_regle(A,B) :- a(A),b(B) { A#1 }. .
{}
?- clause(differents(U,V),  Queue).
{ Queue=true, U#V }
?- clause(regle(U,V),  Queue).
{ Queue=a(U), U#V }
?- clause(longue_regle(U,V),  Queue).
{ Queue=','(a(U), b(V)), U#1 }
```

voir également :

- *rule*

© PrologIA

consult(S)

Fonction	: Insertion de règles
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: *S*, qui doit être une chaîne ou un identificateur
Messages d'erreur	: Fin de fichier inattendue, Impossible d'ouvrir le fichier

Description :

Cette règle prédéfinie permet d'ajouter des énoncés (règles), dans l'ordre dans lequel ils sont lus à partir du fichier *S*. Lorsque *S* est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom de fichier (on ignore donc le préfixe). Lorsque cette chaîne est "user", le fichier de lecture est la console. La lecture se termine lorsque l'on rencontre la fin du fichier ou bien quand une règle vide est lue (une règle vide étant constituée d'un « **.** » suivit d'un **espace**).

Si une erreur de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un «.») sont ignorés, puis l'insertion reprend.

consult provoque une erreur lorsque un paquet lu existe déjà. Dans ce cas, Prolog III sort du « mode insertion », mais ne ferme pas les fichiers qui étaient ouverts : il continue donc à lire le(s) fichier(s) en cours : les faits et les requêtes sont exécutées, les règles provoquent des erreurs de syntaxe .

L'appel *consult(S)* est équivalent à *insert(S)*, lorsque *S* est une chaîne de caractères. *consult("user")* est lui équivalent à *insert* .

Voir également :

- *reconsult, insert*

functor(T, E, N)

Fonction	: Constructeur général d'arbre
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Associe à un arbre *T* son étiquette *E* et son arité *N* . Cette primitive est actuellement décrite par la règle :

```
functor(E[U],E,N) {U::N}.
```

Elle impose à *E* d'être un feuille et à *N* d'être numérique (aucun n'étant forcément connus à l'appel).

Exemples :

```
?- functor(a(b,c), X, N).
{ X=a, N=2 }
?- functor(T, a, 2).
{ T=a[U_1], U_1::2 }
```


© PrologIA

get(X)

Fonction	: Rend le code d'un caractère non blanc lu
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier d'entrée

Description :

X est unifié avec le numéro de code ASCII du premier caractère non blanc lu sur l'entrée courante. La primitive sous-jacente est *in_char'*. La conversion du caractère en code ASCII est faite par la primitive *char_code*.

Voir également :

- *in_char', get0*

get0(X)

Fonction	: Rend le code du prochain caractère lu
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Fin de fichier d'entrée

Description :

X est unifié avec le numéro de code ASCII du prochain caractère lu sur l'entrée courante. La primitive sous-jacente est *in_char* . La conversion du caractère en code ASCII est faite par la primitive *char_code* .

Voir également :

- *in_char, get*

© PrologIA

is(X, T)
X is T

Fonction	: Evaluation d'un arbre
Classe	: Primitive Edimbourg
Catégorie	: Evaluation de fonctions prédéfinies
Paramètres connus	: *T*
Messages d'erreur	: Erreur dans val

Description :

Le terme *T* est évalué, et la valeur obtenue est unifiée à *X* . L'évaluation d'un identificateur donne lui-même à moins qu'il lui ait été auparavant assigné une constante. *is* est implantée à l'aide de la primitive *val* :

```
X is Y  :- val(Y, X).
```

Remarque :

Les opérateurs <, >, =<, >= ne sont pas disponibles dans le membre droit de *is*, il faut utiliser respectivement *inf, sup, infe, supe* en notation fonctionnelle.

Exemples :

```
?- X  is  add(exp(1.0),  1).
{ X=3.7182818284590e+00 }
?-  Y  is  3*4,  X  is  add(2Y/4,  sin(1)).
{ Y=12, X=6.8414709848078e+00 }
?- X is 3*4 < 44/3 .          % mettre un blanc entre 3 et .
Erreur dans val
?- X  is  inf(3,4).
{ X=1 }
```

Voir également :

- *val*

listing
listing(I)
listing(<I, A>)

Fonction	: Impression de règles
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *I* et *A*
Messages d'erreur	: Argument de mauvais type, Règle inexistante

Description :

Ces primitives sont respectivement équivalentes à :
list, list(I), list(<I,A>)

© PrologIA

member(X, L)

Fonction	: Recherche d'un élément dans une liste classique
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

La primitive *member* est décrite par les règles suivantes :

```
member(X, [X|L]).
member(X, [_|L]) :- member(X, L).
```

Exemples :

```
?- member(X,  [1,2,3]).
{ X=1 }
{ X=2 }
{ X=3 }
?- member(3,  [1,2,3]).
{}
?-  member(2,  [1,3,X]).
{ X=2 }
```

name(X, L)

Fonction	: Conversion de constantes en liste de codes de caractères
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Soit *X* , soit *L*
Messages d'erreur	: Ce n'est pas un identificateur

Description :

Si *X* est un identificateur, un nombre, un caractère ou un booléen, *L* est unifié avec la liste des codes ASCII des caractères qui constituent le symbole *X*.

Si *L* est une liste de codes ASCII des lettres d'un identificateur, *X* est instancié avec l'identificateur déterminé par les conventions courantes de préfixages.

Exemples :

```
?- name(12345678,  L).
{ L=[49,50,51,52,53,54,55,56] }
?- name(X,  [97,98,99]).
{ X=abc }
?- name(X,  [65,66]).          %codes Ascii de A et de B
Erreur: Ce n'est pas un identificateur
```

Voir également :

- *string_ident, string_integer, string_real, string_bool, ….*

© PrologIA

nl

Fonction	: Imprime un retour chariot sur la sortie courante
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit un retour chariot sur l'unité de sortie courante . Se comporte comme la primitive *line* .

Voir également :

- *line, outl, outml*

nonvar(T)

Fonction	: Vérifie que *T* a une étiquette connue ou une liste de fils connue.
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

nonvar(T) se comporte comme *not(free(T))*, c'est-à-dire que nonvar réussi si l'étiquette de *T* est connue ou bien si on sait si la liste des fils est vide ou pas. Echoue dans les autres cas. L'appel *nonvar(T)* réussit quand *var(T)* échoue et vice-versa .

Voir également :

- *free, known, bound, var*

© PrologIA

number(R)

Fonction	: Vérifie que *R* est un nombre connu.
Classe	: Primitive Edimbourg
Catégorie	: Vérifications de types
Paramètres connus	: Voir ci-après
Messages d'erreur	: Non

Description :

Cette règle prédéfinie vérifie que *R* est un terme représentant une valeur numérique connue. Si ce n'est pas le cas *number(R)* échoue. *number* est un synonyme de la primitive *num* .

Voir également :

- *num, integer, real, rationnal*

op(N, I1, S)
op(N, I1, S, I2)

Fonction	: Déclaration d'opérateur
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: *N,I1,S,I2*
Messages d'erreur	: Non

Description :

Déclare l'opérateur *S* avec la précédence *N* et le type de parenthésage *I1* . *S* est un soit un identificateur non quoté, soit une chaîne de caractères.

Dans le cas où *S* est une chaîne, son contenu ne doit débuter ni par une constante ni par une variable ni par **espace**. Dans la forme à 3 arguments, *S* peut être une liste, la déclaration s'applique alors à chaque élément de la liste. Le symbole fonctionnel représenté par *S* est *I2* si celui-ci est fourni, sinon le symbole *S* quoté. L'associativité est précisée par un identificateur ou une chaîne de caractères combinant une, deux ou trois des lettres *f,x,y* avec les conventions suivantes :

- *f* représente l'opérateur.
- *x* représente un expression de précédence inférieure à *f*
- *y* représente un expression de précédence inférieure ou égale à *f*

Les combinaisons possibles sont les suivantes : *f fx fy xf yf xfx yfx xfy* . Lorsque *N* est égal à 0, *op* a pour effet de supprimer la déclaration existante pour l'opérateur *S*.

Exemples :

```
?- op(400, xfx, '@@').
{}
?- write(X @@ Y).
'@@'(X,Y){}
```


© PrologIA

put(X)

Fonction	: Envoie sur l'unité courante le code ASCII *X*
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: *X* , qui doit être entier
Messages d'erreur	: Non

Description :

Le code de caractère ASCII *X* est envoyé sur l'unité de sortie courante. Echoue si *X* n'est pas un entier.

Exemples :

```
?- put(65), put(66), put(67), nl.
ABC
{}
```

Voir également :

- *out, outm*

read(T)

Fonction	: Lit un terme sur l'unité courante d'entrée
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Erreurs de syntaxe diverses

Description :

Lit le prochain terme se trouvant sur l'entrée courante. Celui-ci doit se terminer par un point suivit d'un espace (ces caractères sont lus). L'argument *T* est alors unifié avec ce terme. Cette primitive est basé sur *in_term* .

Exemples :

```
?- read(T).
[1,2].
{ T=[1,2] }
```

Voir également :

- *in_term*

© PrologIA

retract(X)

Fonction	: Suppression de règles selon un certain modèle
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: X
Messages d'erreur	: Non

Description :

Supprime toutes les règles qui s'unifient avec *X*. *X* doit être un terme ayant l'une des formes décrites en introduction. La tête de la règle déterminée par *X* doit être un arbre étiqueté par un identificateur connu. Au nombre d'arguments près, *retract(X)* se comporte comme la primitive *retract* d'arité 2 .

Exemples :

```
?- consult("user").
titi(1)  :-  aa.
titi(2)  :-  aa.
titi(3)  :-  aa.
.
?- retract(titi(X)  :-  aa  {X#2}).
{ X=1 }
{ X=3 }
```

Voir également :

- *abolish, suppress, retractall*

retractall(T)

Fonction	: Suppression de règles selon un certain modèle
Classe	: Primitive Edimbourg
Catégorie	: Gestion des règles et des identificateurs
Paramètres connus	: *T*
Messages d'erreur	: Non

Description :

Supprime toutes les règles dont la tête s'unifie avec *T* , lequel n'est donc pas une règle mais une tête de règle. *T* doit être un terme dont l'étiquette est un identificateur connu et dont le nombre de fils est connu.

Exemples :

```
?- consult("user").
titi(1)  :-  aa.
titi(2)  :-  bb.
titi(3)  :-  cc.
.
?- retractall(titi(X))  {X#2}.
{X=1}
{X=3}
```

Voir également :

- *abolish, retract, suppres*s

© PrologIA

see(F)

Fonction	: Définition de l'unité courante d'entrée
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: *F* qui doit être une chaîne ou un identificateur
Messages d'erreur	: Fin de fichier inattendue, Impossible d'ouvrir le fichier

Description :

Se comporte comme *input(F)* si *F* est une chaîne. Dans le cas où *F* est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité passée à *input* (c'est-à-dire que le préfixe est ignoré).

Voir également :

- *input, close_input, seen*

seen

Fonction	: Fermeture de l'unité courante d'entrée
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Impossible de fermer le fichier

Description :

Se comporte comme close_*input*.

Voir également :

- *input, close_input, see*

© PrologIA

tab(N)

Fonction	: Impression de *N* blancs sur la sortie courante
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: *N* qui doit être un entier connu
Messages d'erreur	: Argument de mauvais type

Description :

Envoie *N* blancs sur l'unité courante de sortie. Lorsque *N* n'est pas un entier connu, un message d'erreur est affiché.

Voir également :

- *line, outm, put, writeq*

tell(F)

Fonction	: Définition de l'unité de sortie courante
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: *F*, qui doit être une chaîne ou un identificateur.
Messages d'erreur	: Non

Description :

Se comporte comme *output(F)* si *F* est une chaîne. Dans le cas où *F* est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité passée à la primitive *output* (c'est-à-dire que le préfixe est ignoré).

Voir également :

- *output, close_output, told*

© PrologIA

told

Fonction	: Fermeture de l'unité de sortie courante
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Impossible de fermer le fichier

Description :

Se comporte comme *close_output*.

Voir également :

- *output, close_output, tell*

true

Fonction	: Réussite
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

S'efface toujours avec succès.

Voir également :

- *clause*

© PrologIA

var(T)

Fonction	: Vérifie que *T* est une variable libre
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

var(T) est exécuté avec succès si l'on ne connaît pas l'étiquette initiale de *T* et que l'on ne sait pas si le nombre de ses fils est nul ou non. Echoue dans tous les autres cas. Se comporte comme *free* .

Exemple :

```
?- var(X).
{}
?- var((+X)[U]).
{ X !num, U !tuple }
?- var(+X).
?- var(X(1,2,3)).
?- var(sommet[L]).
?- var(sommet(X,Y,Z)).
?- var(sommet[L]) {L # <>}.
?-
```

Voir également :

- *free, bound, known, nonvar*

write(T)

Fonction	: Impression de termes sans délimiteurs
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Si *T* est un identificateur, une chaîne ou un caractère, *write* écrit *T* sans les délimiteurs. Sinon écrit le terme comme le fait la primitive *out*. Actuellement ne tient pas compte des déclarations d'opérateurs et emploie la notation fonctionnelle.

Exemple :

```
?- write('X').
X{}
?- write("chaine").
chaine{}
?- write(`c`).
c{}
?- write(a('X',1+2X,"str")).
a('X',2X+1,"str"){}
```

Voir également :

- *out, outm, writeq*

© PrologIA

writeq(T)

Fonction	: Impression de termes
Classe	: Primitive Edimbourg
Catégorie	: Entrées/Sorties
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Ecrit le terme *T* sur l'unité courante de sortie, sans les contraintes associées aux variables qui peuvent figurer dans ce terme. *writeq* se comporte comme la primitive *out*. Actuellement *writeq* ne tient pas compte des déclarations d'opérateurs et emploie la notation fonctionnelle.

Exemple :

```
?- writeq('X').
'X'{}
?- writeq("chaine").
"chaine"{}
?- writeq(a('X',1+2X,"str")).
a('X',2X+1,"str"){}
```

Voir également :

- *out, outm, write*

X =.. Y

Fonction	: Eclatement de termes
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: *X* ou *Y*
Messages d'erreur	: Non

Description :

Si l'arité de *X* est connue, *Y* est unifié avec une liste dont le premier élément est l'étiquette de *X*, et les éléments suivants les éventuels arguments dans l'ordre. Sinon, *X* est d'arité inconnue, et *Y* doit être une liste (et non pas une paire pointée) dont le premier élément est atomique, c'est-à-dire une feuille; *X* est alors unifié avec le terme construit ayant le premier élément de la liste comme *foncteur*, et les autres éléments comme arguments. Echoue donc si *X* est d'arité inconnue et *Y* n'est pas une liste, ou bien si *X* et *Y* sont libre (*var*) .

Exemples :

```
?- out(X,a) =.. Y.
{Y=[out,X,a]}
?- X =.. [toto,1,2,3].
{X=toto[<1,2,3>]}                    % X = toto(1,2,3)
?- label[U] =.. [label,1,2|R].
?- X =.. Y.
?-
```

Voir également :

- *functor*

© PrologIA

X < Y X =< Y
X > Y X >= Y

Fonction	: Evaluation et vérifications des relations
Classe	: Primitive Edimbourg
Catégorie	: Evaluation de fonctions prédéfinies
Paramètres connus	: *X* et *Y*
Messages d'erreur	: Erreur dans val

Description :

X et *Y* sont évaluées (voir *val*). Les comparaisons sont ensuite effectuées entre leurs valeurs, qui peuvent être de type numérique, identificateur ou chaîne de caractères. Dans le cas des chaînes, la comparaison est alphabétique, Dans le cas des identificateurs, la comparaison à lieu sur les chaînes composant leur partie abrégée. Ces quatres primitives sont définies comme suit :

```
X<Y      :- val(inf(X,Y), 1).
X=<Y     :- val(infe(X,Y), 1).
X>Y      :- val(sup(X,Y), 1).
X>=Y     :- val(supe(X,Y), 1).
```

Exemples :

```
?- X>3.
Erreur dans val
?- 3*4X > Y  {X=1/4, Y=1}.
{ X=1/4, Y=1 }
?- "ABC" < "BCD".
{}
?- label >= labelle.
?-
```

Voir également :

- *val, is*

X = Y

Fonction	: Unification de deux termes
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Se comporte comme la règle prédéfinies *eq* . Cette primitive est définie comme suit :

```
X = X .
```

Exemples :

```
?- X = 1.
{ X=1 }
?- X = f(X).
{ X=f(X) }
?- f(f(X)) = f(f(f(X))).
{ X=f(X) }
```

Voir également :

- *eq*

© PrologIA

X == Y

Fonction	: Egalité formelle de deux termes
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Réussit quand les termes *X* et *Y* sont formellement égaux, c'est-à-dire quand l'ensemble de solutions de *X* est le même que celui de *Y* . Sinon échec. Cette règle prédéfinie est définie comme suit :

```
X == Y :- not(dif(X,Y)) .
```

Exemples :

```
?- X == 1.
?- X == X.
{}
?- X == Y.
?-
```

Voir également :

- *not, dif*

X \== Y

Fonction	: Non-égalité formelle de deux termes
Classe	: Primitive Edimbourg
Catégorie	: Arbres, listes et tuples
Paramètres connus	: Aucun
Messages d'erreur	: Non

Description :

Réussit quand les termes *X* et *Y* ne sont pas formellement égaux. Cette règle prédéfinie est déclarée comme suit :

```
X \== Y :- not(not(dif(X,Y))) .
```

Exemples :

```
?- X \== 1.
{}
?- X \== X.
?- X \== Y.
{}
```

Voir également :

- *not, dif*

© PrologIA

X =:= Y

Fonction	: Egalité des valeurs de deux termes
Classe	: Primitive Edimbourg
Catégorie	: Evaluation de fonctions prédéfinies
Paramètres connus	: *X* et *Y*
Messages d'erreur	: Erreur dans val

Description :

Evalue les termes *X* et *Y*. Réussit quand la valeur de *X* est égale à la valeur de *Y* , sinon échec. Cette règle prédéfinie est décrite comme ceci :

```
X =:= Y :-  val(eql(X,Y), 1) .
```

Exemples :

```
?- ln(exp(1))  =:=  1  .
{}
?- X  =:=  X.
Erreur dans val
```

Voir également :

- *val, eql, =\=*

X =\= Y

Fonction	: Non-égalité des valeurs de deux termes
Classe	: Primitive Edimbourg
Catégorie	: Evaluation de fonctions prédéfinies
Paramètres connus	: *X* et *Y*
Messages d'erreur	: Erreur dans val

Description :

Evalue les termes *X* et *Y*. Réussi quand la valeur de *X* est différente de la valeur de *Y* , échoue sinon. Cette règle prédéfinie est décrite comme ceci :

```
X =\= Y :- val(eql(X,Y), 0) .
```

Exemples :

```
?- exp(1) =\= ln(1) .
{}
?- X =\= 2.
Erreur dans val
```

Voir également :

- *val, eql, =:=*

© PrologIA

X -> Y

Fonction	: Si-Alors
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: *X* et *Y* qui doivent être connus à leur appel
Messages d'erreur	: Non

Description :

Tente d'effacer *X* une fois. Si c'est possible, exécute alors *Y* de toutes les façons possibles. Voici cette règle prédéfinie :

```
X -> Y  :- X , ! , Y .
```

X , Y

Fonction	: Séquence d'exécution
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: *X* et *Y* qui doivent être connus à leur appel
Messages d'erreur	: Non

Description :

Exécute *X*. Si c'est un succès, exécute alors *Y*. C'est le mécanisme principal d'une queue de règle. Voici cette règle prédéfinie

```
X , Y :- X , Y .
```


© PrologIA

X ; Y

Fonction	: Alternative d'exécution
Classe	: Primitive Edimbourg
Catégorie	: Contrôle
Paramètres connus	: *X* et *Y* qui doivent être connus à leur appel
Messages d'erreur	: Non

Description :

Installe un point de choix. Voici ces règles prédéfinies

```
X ; Y :- X .
X ; Y :- Y .
```

Exemples :

```
?- write(1)  ;  write(2).
1{}
2{}
```

'[]'(X, Y)
[X | Y]
[]

Fonction	: Liste de buts à exécuter
Classe	: Règle prédéfinie Prolog III
Catégorie	: Contrôle
Paramètres connus	: *X* et *Y* qui doivent être connus à leur appel
Messages d'erreur	: Non

Description :

Exécute les buts *X* puis *Y*. Se comporte comme ',' . Cette primitive est décrite comme ceci

```
'[]'.
'[]'(X, Y) :- X, Y .
```

Exemples :

```
?- [out(a),out(b),out(c)].
abc{}
?- [].
{}
```


© PrologIA

Index des règles prédéfinies, procédures externes et fonctions évaluables pour la syntaxe Edimbourg

, 418
-> 417
< 411
= 412
=.. 410
=:= 415
=< 411
== 413
=\= 416
> 411
>= 411
[] 420
\== 414
abolish 377
assert 378
asserta 380
assertz 378
atom 381
atomic 382
call 383
clause 384
consult 385
functor 386
get 387
get0 388
is 389
listing 390
member 391
name 392
nl 393
nonvar 394
number 395
op 396
put 397
read 398
retract 399
retractall 400
see 401
seen 402
tab 403
tell 404
told 405
true 406
var 407
write 408
writeq 409

Association Prolog
HERITAGE

Les syntaxes de Prolog III

1. **Introduction**
   - **Choix du mode syntaxique**
   - **Points communs, différences**
   - **Conventions grammaticales**
   - **Jeu de caractères**
   - **Constantes communes**
2. **Syntaxe de base**
   - **Variables**
   - **Identificateurs**
   - **Termes**
   - **Contraintes**
   - **Règles et requêtes**
   - **Quelques remarques**
3. **Syntaxe d'Edimbourg**
   - **Variables**
   - **Identificateurs**
   - **Termes**
   - **Contraintes**
   - **Règles et requêtes**
   - **Quelques remarques**
4. **Remarques générales**

Que trouve-t-on dans ce chapitre ?

Vous pouvez utiliser Prolog III à travers deux syntaxes différentes, appelées ici « syntaxe de base » *et* « syntaxe Edimbourg » *; leur description est l'objet de ce chapitre.*

© PrologIA

1. Introduction

Choix du mode syntaxique

Au démarrage d'une session Prolog III, la syntaxe utilisée est la syntaxe de base. Vous pouvez en changer en exécutant la commande

```
>   edinburgh;
{}
?-
```

L'apparition du prompt d'Edimbourg, « ?- » vous indique le succès de votre manipulation. Les requêtes peuvent être données immédiatement; par contre, pour l'entrée des règles, il faut utiliser les prédicats prédéfinis *insert* ou *consult*. Vous devez savoir que pour revenir en syntaxe de base, il vous suffit d'exécuter

```
?- prologIII.
{}
>
```

Points communs, différences

Disons brièvement que la syntaxe de base et tirée en grande partie de celle de Prolog II, et que son style est empreint de notations mathématiques. La syntaxe "Edimbourg" est plus répandue dans le monde à travers ses dialectes. En voici un de plus, étendu bien sûr aux fonctionnalités de Prolog III.

© PrologIA

Bien que ces deux syntaxes puissent paraître très différentes, au moins par leur présentation et leur esprit, un travail important de “mise en commun” a été réalisé, permettant, pour un programme Prolog III donné, une traduction très aisée d'une syntaxe vers l'autre. Il était indispensable qu’un lecteur habitué à la syntaxe Edimbourg ne soit pas pénalisé, voire induit en erreur par ses habitudes syntaxiques.

Passons aux points communs et différences. On peut noter comme point commun le jeu de caractères (en première approximation), mais surtout l’ensemble des constantes aux identificateurs près. Il existe toutefois une syntaxe commune d’identificateur, dite d’ “identificateur quoté”, malheureusement peu lisible. Les variables ont des syntaxes distinctes, mais là encore, il existe une forme commune dite de “variable underline[1]”. Insistons sur le fait qu’il existe des moyens de se passer de ces formes alternatives, car les deux syntaxes de variables ont une intersection non-vide et qu’il en va de même pour les identificateurs.

Les termes sous notation fonctionnelle ainsi que les tuples et les listes s’écrivent de façon identique dans les deux syntaxes. Le point de divergence est le moment où, en syntaxe Edimbourg, on décrit sous forme de termes les contraintes, règles et requêtes; cette vision des choses étant grandement favorisée par l’introduction des opérateurs, ces symboles qui permettent des notations infixées, postfixées ou préfixés. Mais si on examine ce qu’on peut produire avec ces syntaxes, on s’aperçoit rapidement que les contraintes s’écrivent de la même façon (à une exception près) et que les règles et requêtes se forment à peu près selon un patron commun.

Conventions grammaticales

Les grammaires de Prolog III sont décrites sous forme d’ensembles de règles hors contexte augmentées de quelques signes spéciaux. Le signe de réécriture est "::=" et le membre gauche d’une règle n’est pas répété lorsqu’il est identique à celui de la règle précédente.

1 Underline est le nom anglo-saxon du caractère _ qu'on surnomme ‘blanc souligné’.

- Les non-terminaux sont des suites de mots entourés par "<" et ">" .
- Les terminaux du langage sont écrits en caractères **gras**. Le mot **espace** désigne quant à lui un caractère parmi les caractères barre d'espacement, touche de tabulation, retour-chariot, et autres caractères non-imprimables[1]. Les blancs qui figurent dans les règles ne sont pas significatifs.
- On appelle élément tout terminal, non-terminal ou toute suite d'éléments parenthésés par les signes "⟦" et "⟧" ou bien par "{" et "}".
- Les crochets "⟦" et "⟧" délimitant une suite d'éléments indiquent que sa présence est facultative.
- Les accolades "{" et "}" indiquent que la suite d'éléments qu'ils parenthèsent peut être absente ou se répéter une ou plusieurs fois.
- La barre " | " désigne une alternative dans une règle et permet de ne pas répéter ce qui précède.
- Un commentaire est un texte en *italique*.

Dans les cas où une règle syntaxique s'exprimerait formellement de manière trop complexe (et serait donc incompréhensible), on se permettra une entorse aux conventions ci-dessus en donnant une règle simplifiée et un commentaire concernant sa validité ou ses exceptions.

Jeu de caractères

Voici le jeu de caractères utilisé par Prolog III :

<jeu de caractères>
    ::= <caractère spécial>
    ::= <lettre>
    ::= <chiffre>

1 Ce sont les caractères dont le code ASCII est compris entre 1 et 32.

© PrologIA

<caractère spécial>
::= **+** | **-** | ***** | **/** | **_** | **!**
::= **(** | **)** | **{** | **}** | **[** | **]** | **<** | **>**
::= **.** | **|** | **&** | **~** | **,** | **;** | **:** | **#**
::= **$** | **'** | **`** | **;**
::= **espace**

<lettre>
::= <minuscule>
::= <majuscule>

<minuscule>
::= **a** | **b** | … | **z** *les accents sont pour plus tard*

<majuscule>
::= **A** | **B** | … | **Z**

<chiffre>
::= **0** | **1** | … | **9**

<chiffre octal>
::= **0** | **1** | … | **7**

<chiffre hexa>
::= **0** | **1** | … | **9** | **A** | … | **F** | **a** | … | **f**

Nous verrons à la fin de ce chapitre, une fois la syntaxe entièrement décrite, le statut particulier du caractère **espace** .

Constantes

Les *constantes* sont les données les plus simples. On distingue les constantes suivantes :

- les identificateurs,
- les nombres entiers non-négatifs, réels non-négatifs,
- les booléens,

- la liste vide,
- les constantes caractères,
- les chaînes de caractères.

<constante>
::= <identificateur>
::= `[]` *liste "classique" vide*
::= <booléen>
::= <entier>
::= <liste vide> *tuple vide*
::= <réel>
::= <constante caractère>
::= <chaîne de caractères>

Voici la syntaxe de ces différentes constantes, à l'exception des identificateurs décrits plus loin :

<entier>
::= <chiffre> { `_` | <chiffre> } *1_123_456 ou 432513*

<booléen>
::= `0' | 1'`

<liste vide>
::= `<>`

<réel>
::= <mantisse> ⟦ <exposant> ⟧ *comme 1.4e-17*

<mantisse>
::= <suite de chiffres> `.` <suite de chiffres>
::= <suite de chiffres> `.`
::= `.` <suite de chiffres>

<exposant>
::= <Ee> ⟦ <signe> ⟧ <suite de chiffres>

© PrologIA

<signe>
    ::= **+** | **-**

<Ee>
    ::= **E** | **e**

<suite de chiffres>
    ::= <chiffre>
    ::= <chiffre> <suite de chiffres>

Pour les réels la syntaxe est simple : au moins un chiffre pour la mantisse, le point y est obligatoire mais l'exposant est facultatif. Attention : le réel lu est le plus long possible, cela peut créer des situations confuses (`2.0f6` se lit « `2 × f6` », où `f6` est une variable, mais `2.0e6` se lit « `2 000 000` »).

Des exemples d'entiers corrects :

```
1234        1_234_567
81591528324789773434561126959611589427200000000
```

Des exemples d'entiers incorrects :

`_27`	*variable*
`12  34`	*suite de deux entiers*
`98.0  0.`	*réels*

Quelques réels valides :

`1.e6`	10^6
`.5e6`	$0{,}5 \times 10^6$
`3.14e-6`	$3{,}14 \times 10^{-6}$
`4.`	*4*
`.666`	*0,666*

Et d'autres qui n'en sont pas :

`e10`	*c'est la variable ou l'identificateur e10*
`.e10`	*la mantisse doit avoir au moins un chiffre*
`1.2e`	*valeur de l'exposant absent*
`3e10`	*absence du point de la mantisse (3×e10)*

Nous allons maintenant introduire le *caractère d'échappement* « \ ». Celui-ci nous permettra, à l'intérieur d'une chaîne ou d'une constante de type caractère, de représenter certains caractères parfois indisponibles au clavier, ou bien d'y masquer certains caractères inopportuns, comme le retour chariot que l'on doit taper pour que le tampon du clavier ne déborde pas lorsqu'on frappe une ligne trop longue.

Nous introduisons ici la définition d'un *caractère désigné*, qui est très proche de ce qui ce passe dans le langage C à ce niveau.

<caractère désigné>
::= **** <lettre sauf **x**>
::= **** <caractère spécial>
::= **** <chiffre octal> <chiffre octal> <chiffre octal>
::= **** <chiffre octal> <chiffre octal>
::= **** <chiffre octal>
::= **\ x** <chiffre hexa> <chiffre hexa>
::= **\ x** <chiffre hexa>

Si le caractère « \ » est suivi :

- d'un retour chariot, celui-ci est ignoré. (aucun caractère)
- d'un entier composé de trois chiffres octaux au plus, le caractère désigné représente le caractère dont le code ASCII est cet entier en base 8.
- de la lettre **x** et d'un entier composé de deux chiffres hexadécimaux au plus, le caractère désigné représente le caractère dont le code ASCII est cet entier en base 16.

© PrologIA

- d'un des caractères du clavier, alors on a la correspondance suivante entre les caractères désignés et les caractères qu'ils représentent :

`\b`	*rubout*	ASCII 8
`\t`	*tabulation*	" 9
`\l`	*linefeed*	" 10
`\f`	*formfeed*	" 11
`\r`	*return*	" 13
`\n`	*newline*	"10 ou 13 *(machine)*[1]
`\e`	*escape*	" 27

Pour tout *c* n'appartenant pas à l'ensemble des caractères `n`, `e`, `f`, `l`, `r`, `t`, `b`, `x`, la séquence `\`*c* désigne le caractère *c* . En particulier `\\` représente `\` , et `\"` le caractère `"`, ce qui permet notamment leur utilisation à l'intérieur des chaînes. Nous pouvons maintenant définir ce que sont une constante caractère et une chaîne de caractères.

<constante caractère>
::= `` ` `` <caractère constant> `` ` `` *il s'agit de l'accent grave*

<chaîne de caractères>
::= `"` { <caractère constant *sauf* `"`> | `""` } `"`

<caractère constant>
::= <caractère du clavier *sauf* retour-chariot>
::= <caractère désigné>

Voici quelques exemples de caractères constants :

`` `a` ``	*représente le caractère a*
`` `\\` ``	*représente le caractère *
`` `\n` ``	*représente le caractère retour à la ligne*
`` `\170` ``	*représente le caractère z*

Quelques exemples de chaînes de caractères :

```
"Toto va a l'ecole"                                    (1)
```

1 Ce code est dépendant de la machine, utilisez\n au lieu de \012 ou \r !

```
"Toto va \
a l'ecole"                                          (2)
"Cette chaine contient\nun retour chariot"          (3)
"Les crochets \"[\" et \"]\" indiquent..."          (4)
```

Les chaînes *(1)* et *(2)* sont identiques; il en est de même pour les chaînes *(3)* et *(4)* . La chaîne *(5)* a pour contenu :

```
Les crochets "[" et "]" indiquent...
```

Les chaînes de caractères suivantes sont incorrectes :

```
"""
"a"b"
```

2. Syntaxe de base

Variables

Les caractères utilisables dans une variable sont les suivants :

<alphanumérique étendu>
::= <lettre>
::= <chiffre>
::= **'** *apostrophe*
::= _ *underline*
::= **$** *dollar*

Dans toutes les modes syntaxiques (syntaxe de base et syntaxe Edimbourg), on peut écrire une variable de la façon suivante :

<variable>
::= _ { <alphanumérique étendu> } *_underline*

par exemple : `_x _1 _toto' _F_F_ _'`

© PrologIA

Un cas particulier de nom de variable est le nom "_", dont chaque occurence représente une nouvelle variable que l'on ne souhaite pas nommer. A cette exception près, cette notation reste exceptionnelle et on lui préfère la forme suivante, spécifique à la syntaxe de base :

<variable>
    ::= <lettre> { <alphanumérique étendu> }

avec la restriction que le deuxième caractère (s'il existe) ne doit pas être une lettre.

Voici des exemples de variables syntaxiquement correctes :

```
x           X           z'          z''
p2c         x78x        m68000
x__y'       u'2
a_
n_ouvelle_variable'  I'can
```

N.B. : L'usage du caractère **$** est déconseillé ; Prolog III l'utilise pour nommer des variables qu'il est amené à créer de manière interne.

Identificateurs

Un *identificateur* peut être complet ou abrégé. Il est complet s'il est préfixé :

<préfixe>
    ::= ⟦ <nom> { **:** <nom> } ⟧ *orbis:lex*

<nom>
    ::= <lettre> { <alphanumérique étendu> }

<identificateur>
    ::= <préfixe> **:** <identificateur abrégé> *sys:outm*
    ::= <identificateur abrégé> *outm*

<identificateur abrégé>
::= <lettre> <lettre> { <alphanumérique étendu> }

On remarquera que la syntaxe des identificateurs abrégés est très proche de celle des variables. La différence à retenir est qu'une variable débute par une lettre au plus alors qu'un identificateur commence par au moins deux lettres.

La forme précédente est spécifique de la syntaxe de base. Dans les deux syntaxes, un identificateur abrégé peut aussi s'écrire :

<identificateur abrégé>
::= **'** { <caractère constant *sauf* **'**> | **''** } **'**

Voici des exemples d'identificateurs corrects :

```
pommme     pommme'
xx'1       xx'2'
zZ44a_45b__c46'
```

en voici d'autres qui sont incorrects (ce sont en fait des variables) :

```
t_iti      I'm_happy            B29
```

Termes

<terme>
::= <terme$_1$>
::= <terme> **<=>** <terme$_1$>
::= <terme> **=>** <terme$_1$>

<terme$_1$>
::= <terme$_2$>
::= **+** <terme$_2$>
::= **-** <terme$_2$>
::= <terme$_1$> **+** <terme$_2$>
::= <terme$_1$> **-** <terme$_2$>
::= <terme$_1$> | <terme$_2$>

© PrologIA

<terme$_2$>
    ::= <terme$_3$>
    ::= <terme$_2$> **&** <terme$_3$>
    ::= <terme$_2$> ***** <terme$_3$>
    ::= <terme$_2$> **/** <terme$_3$>
    ::= <multiplicande> <variable>

<terme$_3$>
    ::= <terme$_4$>
    ::= **~** <terme$_3$>

<terme$_4$>
    ::= <terme$_5$>
    ::= <terme$_5$> **(** <suite de termes> **)**
    ::= <terme$_5$> **[** <terme> **]**
    ::= **[** <suite de termes> **]** *liste "classique"*
    ::= **[** <suite de termes> | <terme> **]** *liste "classique"*
    ::= <terme$_4$> **.** <terme$_5$>

<terme$_5$>
    ::= <variable>
    ::= <constante>
    ::= **<** <suite de termes> **>**
    ::= **(** <terme> **)**

<constante>
    ::= <identificateur>
    ::= **[]** *liste vide*
    ::= <booléen>
    ::= <entier>
    ::= <liste vide> *tuple vide*
    ::= <réel>
    ::= <constante caractère>
    ::= <chaîne de caractères>

<multiplicande>
::= <entier>
::= <réel>
::= (<terme>)

<suite de termes>
::= <terme>
::= <terme> , <suite de termes>

Cette syntaxe va nous permettre de construire des expressions numériques à l'aide en particulier des quatres opérateurs `+ - * /` , des expressions booléennes grâce aux opérateurs booléens `| & ~ <=> =>`, de construire des tuples avec le constructeur de tuple `< >` et l'opérateur de concaténation `.`, de former des arbres à l'aide des parenthèses `()` ou des crochets `[]` .

Expressions numériques

Les expressions numériques de Prolog III sont formées à l'aide de constantes numériques, de variables, des quatre opérations `+-*/` Donnons quelques exemples d'expressions numériques correctes :

```
-x
355/113
-3/2  +  (1000-1*2*3*4*5*6*7)y  +  (1-3/4)*(x+y-2z/5)  +  4
-3x/4  -  (14-22/7)x
(2x-1)y  +  (a-b)/(x-y)
```

et des exemples incorrects :

`5(x+y)`	*est un arbre dont l'étiquette est 5*
`5/4x`	*est syntaxiquement incorrecte*
`xy`	*est un identificateur et non pas x × y*
`3*-x`	*doit s'écrire 3*(-x)*

Expressions booléennes

Les *expressions booléennes* sont formées à l'aide de variables, de booléens et des opérateurs de priorités décroissantes *non*, *et*, *ou*, *équivalent* et *implique*, notés respectivement `~ & | <=> =>`. Le parenthésage permet de modifier les priorités. Voici des exemples d'expressions booléennes correctes :

© PrologIA

```
x
1'
x|y
~~y
x&~x
c&1'
(u3 & ( ~x1 <=> x3 )) | 0'
```

Arbres et Termes

Sur le plan strictement syntaxique, un *terme* est un couple constitué d'une étiquette et d'une liste (éventuellement vide) de fils. Un *arbre* est un terme sans variables. Une *feuille* est un terme sans fils que l'on confond avec son étiquette. Quelques exemples de termes corrects :

```
(ff(x,y))(a,b,c)
(4z+y)(caractere(`c`,  "caractere  `c`"))
<1,2,A,B>.Z
E[<a,b,c>.R]          le constructeur général d'arbre.
[1,2,3  |  L]         une liste "classique".
```

- *On notera que l'on a parenthésé l'étiquette des deux premiers termes .*
- *Dans le dernier exemple la barre est le séparateur de paire-pointée et non pas l'opération ou-booléen. Si on souhaite exprimer la liste contenant l'expression booléenne* **A|B**, *il faut écrire* **[(A|B)]** *au lieu de* **[A|B]**.
- *On insistera fortement sur le fait que syntaxiquement correct ne signifie pas sémantiquement correct.*

Des exemples de termes syntaxiquement incorrects :

```
toto(x,y)(z,t)
toto()
```

Contraintes

Les *contraintes* sont toujours exprimées à l'intérieur d'un ensemble entre accolades { } , appelé *système de contraintes* . On distingue quatre familles de contraintes :

- les contraintes générales d'*égalité* et de *différence* de termes,
- les contraintes de typage de termes,
- les contraintes numériques *inférieur* et *supérieur* (ou égal),
- la contrainte booléenne *implique* .

Voici la syntaxe des systèmes de contraintes :

<système de contraintes>
::= **{ }**
::= **{** <contrainte> {**,** <contrainte>} **}**

<contrainte>
::= <terme> **=** <terme>
::= <terme> **#** <terme>
::= <terme> **=>** <terme>
::= <contrainte de typage>
::= <contrainte Sup>
::= <contrainte Inf>
::= <variable> **!freeze(** <terme> **)**

<contrainte Sup>
::= <terme> <sup> <terme> *a > b*
::= <contrainte Sup> <sup> <terme> *1>a>b>c*

<contrainte Inf>
::= <terme> <inf> <terme>
::= <contrainte Inf> <inf> <terme>

<inf>
::= **<** | **<=**

<sup>
::= **>** | **>=**

Nous pouvons donner la syntaxe des contraintes de typage :

© PrologIA

<type d'arbre>
::= **!idt** | **!boolt** | **!numt** | **!chart** | **!tuple**
::= **!id** | **!bool** | **!num** | **!char**

<contrainte de typage>
::= <terme> <type d'arbre>
::= <terme> **::** <terme>

Les systèmes de contraintes suivants sont équivalents :

`{ t !id }`	*et*	`{ t !idt, t::0 }`
`{ t !bool }`	*et*	`{ t !boolt, t::0 }`
`{ t !num }`	*et*	`{ t !numt, t::0 }`
`{ t !char }`	*et*	`{ t !chart, t::0 }`

Donnons quelques exemples de contraintes correctes :

```
{ x!num, f!idt, f::3, l::7n+u*v }
{ a # b(i,j), "toto"=z, a=>b&c, t < 3x/4+5y }
```

Règles et requêtes

<règle>
::= <tête> **->** { <but> } [**,** <système de contraintes>] **;**

<requête>
::= { <but> } [**,** <système de contraintes>] **;**

<but>
::= **/** | **!** *slash ou cut c'est la même chose*
::= <terme>

<tête>
::= <terme>

avec les restrictions suivantes :

- la simple vérification des contraintes comme elles sont écrites dans la règle doit imposer au terme de <tête> :
  - d'avoir une étiquette qui est un identificateur connu
  - d'avoir un nombre de fils connu
- un but (lorsque c'est un terme) doit être de type *!idt*.
- les buts, lorsqu'ils sont présents, doivent êtres séparés par des caractères **espaces**.
- les règles sont rangées par paquets ayant même nom de prédicat d'accès (identificateur de tête) et par sous-paquets où ces prédicats ont la même arité.

Voici quelques exemples de règles syntaxiquement correctes :

```
repas(x,y,z)  ->  hors_d_oeuvre(x)  plat(y)  dessert(z);
add(a,b,<a,b,c>)  ->  ,{l!tuple,  b!num,  c=a+b*b};
chats_oiseaux_pattes_tetes(c,  o,  4c+2o,  c+o)  ->;
```

quelques exemples de requêtes syntaxiquement correctes :

```
repas(salade,viande,d);
repas(h,sole,d)  {h#salade};
add(a,b,c)  add(c,d,e)  ,{a<0',  e>0};
circuit(c,a,b)  /  ,{c|a&~b  =  456,  d=>c,  d#b};
```

Rappelons qu'une règle ou une requête peut être syntaxiquement correcte sans qu'elle soit pour autant valide et acceptée par le système Prolog III. Les restrictions sémantiques (par le typage par exemple) réduisent fortement les constructions potentielles permises par la syntaxe.

© PrologIA

Quelques remarques

Caractère espace.

Formulons quelques remarques sur le statut du **espace**. Il peut être inséré partout sans effet de bord, sauf à l'intérieur :

- d'une variable,
- d'une constante,
- d'un but de la forme `tt(gg)` , où un espace entre `tt` et la parenthèse créerait une scission en deux buts `tt` et `(gg)` , ou une erreur de syntaxe.

De même, **espace** peut être enlevé de partout sauf à l'intérieur d'une constante et aux endroits où son retrait provoque la fusion de deux unités lexicales en une seule.

Continuation de ligne.

La séquence \<retour-chariot> est complétement ignorée pendant la lecture de terme et de règles. Cela résout des problèmes de présentation (chaine ou nombre trop long). Par exemple,

```
regle(123456\
78912)  ->;
```

(le 7 étant bien en début de la seconde ligne) est la même chose que :

```
regle(12345678912)  ->;
```

Les différents commentaires

Un *commentaire* est une suite de caractères parenthésés par les groupements particuliers `/*` et `*/` . Il peut apparaître partout où un caractère **`espace`** serait non-significatif. On peut aussi parenthéser les commentaires à l'aide de groupements formés par la répétition de `/*` pour l'ouverture et de `*/` pour la fermeture, avec un même facteur de répétition. Ceci permet d'emboîter les commentaires. Voici des exemples :

```
/* ceci est un
commentaire tenant sur deux lignes*/

/*/* le texte /* commentaire */ est inclus */*/
```

Un autre type de commentaire est celui introduit par le caractère `%` et qui se termine à la fin de la ligne courante.

```
texte à commenter              % commentaire
```

3. Syntaxe d'Edimbourg

Variables

Les caractères utilisables dans une variable sont les suivants :

<alphanumérique étendu>
::= <lettre>
::= <chiffre>
::= **'** *apostrophe*
::= **_** *underline*
::= **$** *dollar*

Dans toutes les modes syntaxiques (syntaxe de base et syntaxe Edimbourg), on peut écrire une variable de la façon suivante :

© PrologIA

<variable>
::= _ { <alphanumérique étendu> } *_underline*

par exemple : `_x _1 _toto' _F_F_ _'`

Un cas particulier de nom de variable est le nom "_", dont chaque occurence représente une nouvelle variable que l'on ne souhaite pas nommer. A cette exception près, cette notation reste exceptionnelle et on lui préfère la forme suivante, spécifique à la syntaxe Edimbourg :

<variable>
::= <majuscule> { <alphanumérique étendu> }

Voici des exemples de variables syntaxiquement correctes :

```
X          W'         Z''
P2c        Nouvelle_variable
X__y'      U'2
A_         I'can
```

N.B. : L'usage du caractère `$` est déconseillé ; Prolog III l'utilise pour nommer des variables qu'il est amené à créer de manière interne.

Identificateurs

Un *identificateur* peut être complet ou abrégé. Il est complet s'il est préfixé :

<préfixe>
::= ⟦ <nom> { `:` <nom> } ⟧ *orbis:lex*

<nom>
::= <lettre> { <alphanumérique étendu> }

<identificateur>
::= <préfixe> `:` <identificateur abrégé> *sys:outm*
::= <identificateur abrégé> *outm*

<identificateur abrégé>
::= <minuscule> { <alphanumérique étendu> }

On remarquera que la syntaxe des identificateurs abrégés est très proche de celle des variables. La différence à retenir est qu'une variable débute par une lettre majuscule alors qu'un identificateur commence par une lettre minuscule.

La forme précédente est spécifique de la syntaxe Edimbourg. Dans les deux syntaxes, un identificateur abrégé peut aussi s'écrire entre simples guillemets :

<identificateur abrégé>
::= ' { <caractère constant *sauf* '> | '' } '

Voici des exemples d'identificateurs corrects :

```
pommme     pommme'    a
x'1        xx'2'      a_
zZ44a_45b__c46'
```

en voici d'autres qui sont incorrects (ce sont en fait des variables) :

```
Titi       I'm_happy            B29
```

+ *Les identificateurs entre guillemets ne sont pas des chaines de caractères.*

Termes

Opérateurs

Les opérateurs permettent d'exprimer de façon simples les arbres et expressions. Leur syntaxe est la suivante :

<opérateur>
::= <constante>
::= <variable>
::= <suite de caractères>

avec la restriction que <suite de caractères> *ne commence ni par une constante ni par une variable ni par un* **espace** *ni par % ni par /* .*

La longueur de cette suite dépend des déclarations d'opérateurs connues de Prolog III au moment de la lecture. Cette suite est la plus longue possible parmi ces déclarations.

On distingue quatre sortes d'opérateurs : ceux qui sont préfixés et admettent un opérande à droite, ceux postfixés qui ont un opérande à gauche, les opérateurs infixes ayant deux opérandes (à droite et à gauche) et ceux qui sont isolés (sans opérandes).

Nous noterons ces types d'opérateurs *fz zf zfz f* qui résument la façon dont l'expression est construite (*f* désignant l'opérateur et le ou les *z* désignant les opérandes). On attribue à tout opérateur sa priorité (ou précédence) sous la forme d'un nombre entier *n*. Par convention, la précédence des opérandes se calcule à l'aide de cet *n* et de la forme exacte du type de l'opérateur (*xf*, *yf*, *.f* , …) de la façon suivante : si en lieu d'un opérande (*z*) on trouve *x* la précédence est *n-1*, si on trouve *y* la précédence est *n*, sinon on trouve un '.' et la précédence est un nombre indépendant de *n* et fourni en même temps que l'opérateur. Indiquons de suite que la forme où l'un des opérandes est un point (*.fx*, …) ne sert que pour construire des termes composés[1]. Voici un tableau récapitulatif du calcul des précédences :

type	f	xf	yf	fx	fy	xfx	xfy	yfx
précédence op.	n	n	n	n	n	n	n	n
préc. op. gauche (*g*)		n-1	n			n-1	n-1	n
préc. op. droite (*d*)				n-1	n	n-1	n	n-1

1 Le prédicat prédéfini *op* ne permet pas de définir de tels opérateurs composés.

(*suite*)

type	.f	f.	xf.	.fx	yf.	.fy
précédence op.	n	n	n	n	n	n
précédence fournie	m	m	m	m	m	m
préc. op. gauche (*g*)	m		n-1	m	n	m
préc. op. droite (*d*)		m	m	n-1	m	m

<f, *n*>
::= <opérateur>
<fz, *n*, *d*>
::= <opérateur>
<zf, *n*, *g*>
::= <opérateur>
<zfz, *n*,*g*,*d*>
::= <opérateur>

Les termes

<termes:, *n*>
::= <fz, *n*,*d*> <terme, *d*> *+x*
::= <terme, *g*> <zfz, *n*,*g*,*d*> <terme, *d*> *a*b*
::= <terme, *g*> <zf, *n*,*g*> *x--*
::= <f, *n*> *18*
::= <terme, *n-1*>
::= <terme composé, *n*>

<terme composé, *n*>
::= <f. ,*n*,*M*> <terme, *M*> <.f,*n*,*M*> *(x)*
::= <f. ,*n*,*M*> <terme,*M*> <.fz,*n*,*M*,*d*> <terme, *d*> *(2+1)x*
::= <terme, *g*> <zf. , *n*,*g*,*M*> <terme, *M*> <.f, *n*,*M*> *f[x]*

Cette dernière règle <terme composée, *n*> permet de décrire sous forme d'opérateurs les divers mécanismes de parenthésage par des symboles comme les parenthèses, crochets, chevrons, et tout couple d'opérateurs simples que l'on souhaite mettre en correspondance.

Nous allons maintenant décrire les opérateurs à travers un tableau dont le contenu ligne par ligne a la forme suivante :

© PrologIA

- les caractères composant l'opérateur (sauf s'il s'agit d'une famille d'opérateurs comme *entier*, en italique)
- un nom interne qui nous servira à préciser ce qu'on a construit.
- le type de l'opérateur .
- la précédence de l'opérateur.
- la précédence fournie (dans le cas où l'opérateur est destiné a être appairé à un autre; ils doivent dans ce cas avoir une même précédence fournie).
- une action parmi {empile, dépile, teste, teste non} qui permet de manipuler et tester le contexte[1] courant. Il faut que l'action réussisse pour que l'opérateur soit applicable. Ces actions ont pour paramètre un caractère; voici leurs effets respectifs :

  - empile *car* établit le nouveau contexte *car* .
  - dépile *car* teste que le contexte est bien *car* et restitue le précédent contexte.
  - teste *car* teste que le contexte est bien *car* .
  - teste non *car* teste que le contexte n'est PAS *car* .

  Une action réussi lorsque le test est vérifié (empile réussi toujours).

Opérateur	Nom Interne	Type	Préc. op	Préc. bis	Action
{	CONTR	xf.	1300	10001	*empile* '{'
{	CONTR	f.	1300	10001	*empile* '{'
}	CONTR	.f	1300	10001	*dépile* '}'
:-	FLECHE	xfx	1200		
;	PVIRG	xfy	1100		
,	VLISTE	xfy	1000		*teste* '['
,	VFONCT	xfy	1010		*teste* 'f'
,	VTUPLE	xfy	1010		*teste* '<'
,	VCONTR	xfy	1010		*teste* '{'
,	VIRG	xfy	1010		
!	CUT	f	700		
=	EGAL	xfx	650		
#	DIFF	xfx	650		
=>	IMPLIQUE	xfx	650		

1 On appelle ici *contexte* le sommet d'une pile de caractère (vide au départ).

::	DEUXPTS	xfx	650		
!tuple	TYPTUPLE	xf	650		
!boolt	TYPBOOLT	xf	650		
!bool	TYPBOOL	xf	650		
!numt	TYPNUMT	xf	650		
!num	TYPNUM	xf	650		
!chart	TYPCHART	xf	650		
!char	TYPCHAR	xf	650		
!idt	TYPIDT	xf	650		
!id	TYPID	xf	650		
!freeze(	TYPFREEZE	xf.	650	10002	*empile* 'z'
)	TYPFREEZE	.f	650	10002	*dépile* 'z'
Opérateur	Nom Interne	Type	Préc. op	Préc. bis	Action
<	INF	xfx	650		
=<	INFEG	xfx	650		
>	SUP	xfx	650		
>=	SUPEG	xfx	650		
<=>	EQUIV	yfx	600		
|	BARRE	xfx	1002		*teste* '['
|	OU	yfx	500		*teste non* '['
+	ADD2	yfx	500		
-	SUB2	yfx	500		
&	ET	yfx	400		
*	MULT	yfx	400		
/	DIV	yfx	400		
+	ADD1	fy	200		
-	SUB1	fy	200		
~	NON	fy	200		
.	CONC	yfx	100		
<	TUPLE	f.	50	10003	*empile* '<'
>	TUPLE	.f	50	10003	*dépile* '<'
[	GENERAL	xf.	50	10004	*empile* 'c'
]	GENERAL	.f	50	10004	*dépile* 'c'
[	LISTE_ED	f.	50	10005	*empile* '['
]	LISTE_ED	.f	50	10005	*dépile* '['
(	FONCTION	xf.	50	10006	*empile* 'f'
)	FONCTION	.f	50	10006	*dépile* 'f'
(	PARENTH	f.	2	10007	*empile* '('
)	PARENTH	.fx	2	10007	*dépile* '('
)	PARENTH	.f	2	10007	*dépile* '('
entier	NUMVAR	fx	2		
entier	ENTIER	f	2		

© PrologIA

ident.	IDENT	f	2
[ ]	IDENT	f	2
carac.	CHAR	f	2
booléen	BOOL	f	2
chaîne	CHAINE	f	2
<>	VIDE	f	2
flottant	NUMVAR	f x	2
flottant	FLOAT	f	2
variable	VAR	f	1

+ *On peut constater que certains opérateurs se retrouvent sur deux lignes avec des types ou précédences différentes.*
Par exemple, entier *peut être un opérateur préfixé (produit par défaut d'un nombre et d'une variable en l'occurence).*

Ce tableau étant quelque peu abrupt, nous allons aussi présenter la syntaxe des termes sous une forme plus habituelle dans laquelle les opérateurs composés et opérateurs isolés (*f*) n'apparaissent plus en tant que tels, mais forment le corps des règles de grammaire.

<terme>
::= <terme, 1300>

<terme, 1300>
::= <terme, 1299> ⟦ <système de contraintes> ⟧
::= <système de contraintes>

<système de contraintes>
::= **{** <terme, 999> {**,** <terme, 999>} **}**

<terme, *n*>
::= <fz, *n*,*d*> <terme, *d*> *+x*
::= <terme, *g*> <zfz, *n*,*g*,*d*> <terme, *d*> *a*b*
::= <terme, *g*> <zf, *n*,*g*> *x--*
::= <terme, *n-1*>

<terme, 50>
::= <terme, 49>
::= <terme, 49> **(** <suite de termes> **)** *fonction*
::= <terme, 49> **[** <terme> **]** *constructeur d'arbre*
::= **[** <suite de termes> **]** *liste "classique"*
::= **[** <suite de termes> | <terme, 999> **]** *paire pointée*
::= <terme, 50> **.** <terme, 49> *concaténation*

<terme, 2>
::= <variable>
::= <constante>
::= **<** <suite de termes> **>** *tuple*
::= <multiplicande> <variable> *produit par défaut*
::= **(** <terme, 1300> **)**

<constante>
::= <identificateur>
::= **[]** *liste vide*
::= <booléen>
::= <entier>
::= <liste vide> *tuple vide*
::= <réel>
::= <constante caractère>
::= <chaîne de caractères>

<multiplicande>
::= <entier>
::= <réel>
::= **(** <terme, 1300> **)**

<suite de termes>
::= <terme, 999>
::= <terme, 999> **,** <suite de termes, 999>

© PrologIA

Cette syntaxe va nous permettre de construire des expressions numériques à l'aide en particulier des quatres opérateurs `+ - * /` , des expressions booléennes grâce aux opérateurs booléens `| & ~ <=> =>`, de construire des tuples avec le constructeur de tuple `< >` et l'opérateur de concaténation `•`, de former des arbres à l'aide des parenthèses `()` ou des crochets `[]` .

Expressions numériques

Les expressions numériques autorisées par Prolog III sont formées à l'aide de constantes numériques, de variables, des quatre opérations `+-*/` Ces expressions représentent des nombres. Donnons quelques exemples d'expressions numériques correctes :

```
-X
355/113
-3/2  +  (1000-1*2*3*4*5*6*7)Y  +  (1-3/4)*(X+Y-2Z/5)  +  4
-3X/4  -  (14-22/7)X
(2X-1)Y  +  (A-B)/(X-Y)
```

+ *Insistons sur le fait qu'une expression numérique reste une feuille et ne forme donc pas un arbre de profondeur supérieure à zéro.*

et des exemples incorrects :

`5(X+Y)`	*est un arbre dont l'étiquette est 5*
`5/4X`	*est syntaxiquement incorrecte*
`XY`	*est bien sûr une variable et non pas $X \times Y$*

Expressions booléennes

Les *expressions booléennes* sont formées à l'aide de variables, de booléens et des opérateurs de priorités décroissantes *non*, *et*, *ou*, *équivalent* et *implique,* notés respectivement `~ & | <=> =>` . Le parenthésage permet de modifier les priorités. Toute expression booléenne représente un booléen (connu ou pas). Voici des exemples d'expressions booléennes correctes :

```
X          1'          X|Y
~~Y        X&~X
```

```
C&1'  =>  D
(U3  &  (  ~X1  <=>  X3  ))  |  0'
```

+ *Ici encore, toute expression booléenne reste une feuille et ne forme donc pas un arbre de profondeur supérieure à zéro.*

Arbres et Termes

Sur le plan strictement syntaxique, un *terme* est un couple constitué d'une étiquette et d'une liste (éventuellement vide) de fils. Un *arbre* est un terme sans variables. Une *feuille* est un terme sans fils que l'on confond avec son étiquette. Voici quelques exemples de termes corrects :

```
(ff(X,Y))(A,B,C)
(4Z+Y)(caractere(`c`,  "caractere  `c`"))
<1,2,A,B>.Z
a(B,C)  :-  b,  c  {B=1}
{A<B,  C  =  <1,2>}
E[<a,b,c>.R]          le constructeur général d'arbre.
[1,2,3  |  L]         une liste "classique".
```

- *On notera que l'on a parenthésé l'étiquette des deux premiers termes .*
- *Dans le dernier exemple la barre est le séparateur de paire-pointée et non pas l'opération ou-booléen. Si on souhaite exprimer la liste contenant l'expression booléenne* `A|B`*, il faut écrire* `[(A|B)]` *au lieu de* `[A|B]`*.*
- *On insistera fortement sur le fait que syntaxiquement correct ne signifie pas sémantiquement correct.*

Des exemples de termes syntaxiquement incorrects :

```
toto(x,y)(z,t)
E[1,2]
(1,2)X
toto()
{C=<1,2>}             mettre des blancs autour du =
```


© PrologIA

Contraintes

Les *contraintes* sont toujours exprimées à l'intérieur d'un ensemble entre accolades { } , appelé *système de contraintes* . On distingue quatre familles de contraintes :

- les contraintes générales d'*égalité* et de *différence* de termes,
- les contraintes numériques *inférieur* et *supérieur* (ou égal),
- la contrainte booléenne *implique* ,
- les contraintes de typage de termes,

Bien qu'un système de contraintes puisse s'exprimer sous forme de terme, nous allons en donner un apercu plus "ciblée" de sa syntaxe.

<système de contraintes>
::= **{** <contrainte> {**,** <contrainte>} **}**

<contrainte>
::= <terme, 650>

<terme, 650>

::= <terme, 649> **=** <terme, 649>		*égalité générale*
::= <terme, 649> **#** <terme, 649>		*différence générale*
::= <terme, 649> **=>** <terme, 649>		*implique*
::= <terme, 649> **<** <terme, 649>		*inférieur*
::= <terme, 649> **=<** <terme, 649>		*inférieur ou égal*
::= <terme, 649> **>** <terme, 649>		*supérieur*
::= <terme, 649> **>=** <terme, 649>		*supérieur ou égal*
::= <terme, 649> **::** <terme, 649>		*contrainte de taille*
::= <terme, 649> **!tuple**		*typage*
::= <terme, 649> **!boolt**		*typage*
::= <terme, 649> **!bool**		*typage*
::= <terme, 649> **!numt**		*typage*
::= <terme, 649> **!num**		*typage*
::= <terme, 649> **!chart**		*typage*
::= <terme, 649> **!char**		*typage*
::= <terme, 649> **!idt**		*typage*
::= <terme, 649> **!id**		*typage*
::= <terme, 649> **!freeze(** <terme> **)**		*gel*

Les systèmes de contraintes suivants sont équivalents :

```
{ t !id }          et        { t !idt,    t::0  }
{ t !bool }        et        { t !boolt,  t::0  }
{ t !num }         et        { t !numt,   t::0  }
{ t !char }        et        { t !chart,  t::0  }
```

Donnons quelques exemples de contraintes correctes :

```
{ X!num, F!idt, F::3, L::7N+U*V }
{ A # B(I,J), "toto"=Z, A=>B&C, T < 3X/4+5Y }
```

Voici quelques exemples de contraintes invalides :

```
{ X<Y<Z }          on ne peut pas emboiter les <
{ ff(1,2) }        ce n'est pas une contrainte
```

Dans les cas où les contraintes n'agissent pas, (quand elles sont en position de sous-terme par exemple), celle-ci se comportent comme tout arbre. Il en va de même avec les systèmes. Voici alors ce qui est produit :

• t1 = t2	construit l'arbre	sys:'='(t1, t2)
• t1 # t2	"	sys:'#'(t1, t2)
• t1 < t2	"	sys:'<'(t1, t2)
• t1 =< t2	construit l'arbre	sys:'=<'(t1, t2)
• t1 > t2	"	sys:'>'(t1, t2)
• t1 >= t2	"	sys:'>='(t1, t2)
• t1 !tuple	"	sys:'!tuple'(t1)
• t1 !boolt	"	sys:'!boolt'(t1)
• …….…….	"	……..…….…….….
• t1 !freeze(t2)	"	sys:'!freeze'(t1, t2)
• t1 {t2}	"	sys:'{}'(t1, t2)
• {t1}	"	sys:'{}'(t1)
• {t1, t2}	"	sys:'{}'(sys:','(t1, t2))
• t1 => t2	calcule le terme	t1 => t2

+ *Notez bien la dernière ligne qui représente un exception dans le mécanisme général de construction. En effet* implique *est une relation dans une contrainte et une opération partout ailleurs.*

Règles et requêtes

<règle>
::= <terme> <stop>

<requête[1]>
::= ⟦ **?-** ⟧ <terme> <stop>

<stop>
::= **. espace**

De façon plus pragmatique, règle et requête ont la forme suivante :

<règle>
::= <terme> ⟦ <système de contraintes> ⟧ <stop>
::= <terme> **:-** <queue> ⟦ <système de contraintes> ⟧ <stop>

<requête>
::= ⟦ **?-** ⟧ ⟦ <queue> ⟧ ⟦ <système de contraintes> ⟧ <stop>

<queue>
::= <terme>
::= <terme> **,** <queue>

Sur toute règle ou requête portent les restrictions suivantes :

- la simple vérification des contraintes comme elles sont écrites dans la règle doit imposer au terme de tête (s'il existe) :
  - d'avoir une étiquette qui est un identificateur connu
  - d'avoir un nombre de fils connu
- un but (lorsque c'est un terme) doit être de type *!idt*.
- les règles sont rangées par paquets ayant même nom de prédicat d'accès (identificateur de tête) et par sous-paquets où ces prédicats ont la même arité.

1 Dans cette implémentation, `:-` ne peut servir à débuter une requête.

© PrologIA

Il est important de noter que les seules contraintes qui vont “agir” sont celles en position indiquée par les règles qui précèdent. En effet, à l'intérieur d'un terme (donc en position de sous-terme), elles se comportent comme un arbre. Il en va de même avec les opérateurs formant les règles (comme ‘ : -’ , ‘ , ’ et ‘ ; ’). Dans ce cas les termes construits sont les suivants :

• t1 :- t2	construit l'arbre	sys:':-'(t1, t2)
• t1 , t2	"	sys:','(t1, t2)
• t1 ; t2	"	sys:';'(t1, t2)

Voici quelques exemples de règles syntaxiquement correctes :

```
repas(X,Y,Z)  :-  hors_d_oeuvre(X),  plat(Y),  dessert(Z).
add(a,B,<A,B,B>)  {L!tuple,  B!num,  C=A+B*B}.
chats_oiseaux_pattes_tetes(C,  O,  4C+2*O,  c+O).
```

quelques exemples de requêtes syntaxiquement correctes :

```
repas(salade,viande,D).
repas(H,sole,D)  {H#salade}.
?-  sub(A,B,C),  add(C,D,E)  {A<0',  E>0}.
circuit'(B,  A,B),  !  {C|A&~B  =  456,  D=>C,  D#B}.
```

Rappelons qu’une règle ou une requête peut être syntaxiquement correcte sans qu’elle soit pour autant valide et acceptée par le système Prolog III. Les restrictions sémantiques (par le typage par exemple) réduisent fortement les constructions potentielles permises par la syntaxe.

Quelques remarques

Caractère espace.

Formulons quelques remarques sur le statut du caractère **espace**. Il peut être inséré partout sans effet de bord, sauf à l’intérieur d’une variable ou d’une constante.

De même, **espace** peut être enlevé de partout sauf à l'intérieur d'une constante et aux endroits où son retrait provoque la fusion de deux unités lexicales en une seule (problème typique des opérateurs).

Un **espace** doit suivre impérativement le caractère . (point) de fin de règle ou de requête. C'est ainsi qu'on distingue ce point final de celui de la concaténation.

Continuation de ligne.

La séquence \<retour-chariot> est complétement ignorée pendant la lecture de terme et de règles. Cela résoud des problèmes d'affichage (chaine ou nombre trop long). Par exemple,

```
regle(123456\
78912)  ->;
```

(le 7 étant bien en début de la seconde ligne) est la même chose que :

```
regle(12345678912)  ->;
```

Les différents commentaires

Un *commentaire* est une suite de caractères parenthésés par les groupements particuliers /* et */ . Il peut apparaître partout où un caractère **espace** serait non-significatif. On peut aussi parenthéser les commentaires à l'aide de groupements formés par la répétition de /* pour l'ouverture et de */ pour la fermeture, avec un même facteur de répétition. Ceci permet d'emboîter les commentaires. Voici des exemples :

```
/* ceci est un
commentaire tenant sur deux lignes*/

/*/* le texte /* commentaire */ est inclus */*/
```

Un autre type de commentaire est celui introduit par le caractère % et qui se termine à la fin de la ligne courante.

```
texte à commenter              % commentaire
```


© PrologIA

Oopérateurs Edimbourg

D'autres opérateurs "Edimbourg" ne servant pas directement à Prolog III sont déclarés dans cette syntaxe. En voici la liste et ce qui est construit :

Opérateur	Type	Précédence	Construction
is	xfx	700	sys:is(t1,t2)
==	xfx	700	sys:'=='(t1,t2)
=\=	xfx	700	sys:'=\='(t1,t2)
=:=	xfx	700	sys:'=:='(t1,t2)
\==	xfx	700	sys:'\=='(t1,t2)
->	xfx	700	sys:'->'(t1,t2)
=..	xfx	700	sys:'=..'(t1,t2)

Etrangetés

Le terme **'+'(X,Y)** provoque une erreur de syntaxe (conflit de type); en effet, ce terme est équivalent à **+(X,Y)** qui est l'opération *+unaire* appliquée au terme **(X,Y)** lequel ne peut être numérique. Si on souhaite contruire le terme étiqueté par '+' et ayant deux arguments X et Y, il faut écrire **('+')(X,Y)** .

Le but suivant sera mal compris par le système :

```
?- X is 3*4.
```

En effet, le nombre **4** suivit de **.** est compris en tant que nombre flottant et le système reste en attente de la suite du but (n'ayant pas vu le stop). Il faut mettre un blanc entre le **4** et le point final.

L'écriture des termes avec opérateurs ne se fait pas au mieux dans cette implémentation. Ainsi la notation fonctionnelle est utilisée pour les opérateurs qui ne sont pas ceux manipulés au cœur de Prolog III.

```
?- out(X is 3*4).
is(X,12).
```

```
{}
```

4. Remarques générales

Différences des modes

Terminons ce chapitre par quelques différences entre ces deux modes syntaxiques.

- Les opérateurs + et – unaires ne sont pas de même priorités dans ces deux syntaxes. En syntaxe de base ils sont au même niveau que les + et – binaires .
- Il n'est pas possible en syntaxe Edimbourg d'écrire une cascade de contraintes *inférieur* (resp. *supérieur*).
- Le programmeur ne peut pas définir d'opérateur dans la syntaxe de base. (en syntaxe Edimbourg, voir le prédicat *op*)
- En syntaxe Edimbourg, le => (*implique*) a une priorité inférieur à celle de <=> (*équivalent*), alors qu'elles sont égales en syntaxe de base.
- En syntaxe de base, il est possible de rentrer directement des règles. En mode Edimbourg, certaines règles (*faits*) ressemblants syntaxiquement à des requêtes, il était nécessaire de posséder un mode *requête* et un mode *insertion de règle* (voir les prédicats *insert* et *consult*) .

Cohabitation des deux syntaxes

Un point gênant la cohabitation de règles dans les deux syntaxes est celui de l'affichage de termes provenant de l'autre syntaxe et contenant des variables. Un exemple sera plus explicite :

```
> egal(x,x)  ->;
> edinburgh;
{}
?- list(egal).
egal(x,x).                  % x n'a pas ici la syntaxe de variable
{}                          % bien qu'elle SOIT une variable
?- egal(1,B).
{B=1}
?- assert(titi(Var,Var)).
```


© PrologIA

```
{}
?- prologIII.
{}
> list(titi);
titi(Var, Var) ->;          % Var a l'air ici d un identificateur
{}                          % mais  c'est une variable
> titi(11, X);
{ X = 11 }
>
```

Ceci n'est pas gênant tant qu'on ne demande pas d'écriture de terme (par solution ou par les prédicats prédéfinies *trace*, *out*, *write*, …). Dans ces cas, les termes écrits peuvent sembler confus. Indiquons que dans cette version, *assert* utilise l'écriture….

Association Prolog
HERITAGE

Primitives Graphiques

1. **Introduction et conventions**
2. **Primitives de gestion des fenêtres**
3. **Dessin et positionnement**
4. **Usage de la souris**
5. **Modes de dessin et d'écriture**
6. **Primitives spéciales de lecture et d'écriture**
7. **Primitive de description de menus**
8. **Gestion des boîtes de dialogue**
9. **Boutons actifs**
10. **Primitives de manipulation de fichiers**

Que trouve-t-on dans ce chapitre ?

Le système Prolog III comporte un dispositif asynchrone de gestion des événements permettant le fonctionnement de l'environnement (fenêtrage, menus, ...) sans que l'utilisateur ait à réorganiser son programme sous la forme de boucle de gestion des événements. La plupart des événements tels que le redimensionnement d'une fenêtre, le défilement des fenêtres de texte ..., sont gérés automatiquement. L'utilisateur peut associer des procédures Prolog à certains événements comme la sélection d'un menu, ou le rafraîchissement d'une fenêtre graphique. Les événements de type position de la souris ou réalisation d'un clic peuvent être testés par programme, ce qui permet de réaliser des boucles d'événements souris.

© PrologIA

1. Introduction et conventions

Ce chapitre décrit une bibliothèque des primitives mises à la disposition du programmeur Prolog III pour la réalisation d'interfaces graphiques. Il suppose du lecteur une certaine pratique des environnements graphiques multi-fenêtres et de leurs boîtes à outils.

Sous Unix ou VMS, on suppose qu'on est placé dans un environnement de type XWindows (X11) ou SunView. Il faut avoir au préalable lancé Prolog III avec l'option *-W* correspondant à l'environnement graphique disponible, qui sont *-WS* pour SunView, *-WX* pour XWindows.
Sur ces machines, le graphique est initialisé par la commande Prolog III :

```
> load_graphic ;
```

Le terme *"console"* désigne la fenêtre de lancement de Prolog III (celle où apparaît la bannière PrologIA).
Les fenêtres graphiques que l'on peut ouvrir à partir de Prolog III sont de deux types : *"GRAPHICS"* ou *"MODAL"*.

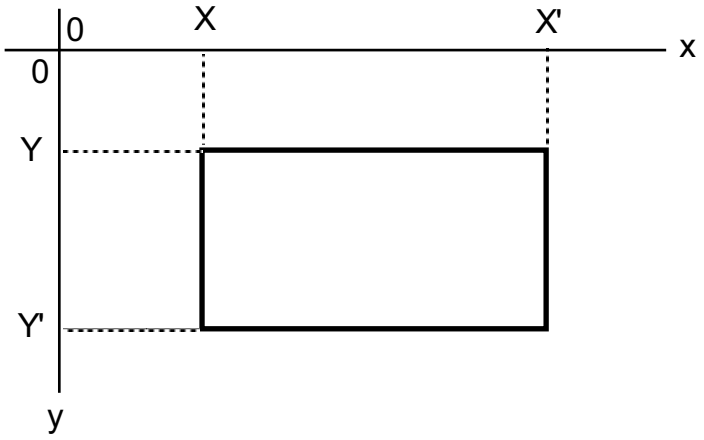
Les objets attachés à une fenêtre peuvent être de nature variée : champs éditables, boîtes de choix, boutons (actifs ou non), menus de type *"pop-up"*, textes et dessins, etc. Les boîtes de dialogue (cf. ***gr_dialog***) utilisent en particulier des fenêtres de type *"MODAL"* , ce qui a pour effet de rendre la saisie obligatoire dans cette fenêtre.

Tous les effets graphiques obtenus ici doivent être considérés, du point de vue de Prolog, comme des effets de bord. En particulier, un objet dessiné n'est jamais effacé au backtracking. De même, la position du point d'écriture est une variable globale.

© PrologIA

Par défaut, l'image *"bitmap"* d'une fenêtre n'est pas sauvegardée, ce qui veut dire que le rafraîchissement n'est pas assuré. Il est possible de le forcer par l'option <*SAVE*> des primitives ***new_window*** et ***set_window***.

On représentera un point de coordonnées d'abscisse *X* et d'ordonnée *Y* sous la forme d'un doublet de nombres <*X, Y*> . De même, un rectangle sera donné sous la forme d'un quadruplet de nombres[1] <*X,Y, X',Y'*>, correspondant au schéma suivant :



On représentera un paquet de règles de nom *Identificateur* et d'arité *arité* sous la forme d'un doublet <*Identificateur, arité*> .

Enfin, par convention, les primitives qui n'ont d'effet que dans une fenêtre graphiques sont préfixées par ***gr_*** . Ces règles utilisent directement les procédures du système graphique de la machine sous-jacente.

Un signe dans la marge indique une primitive ou une caractéristique spécifique au Macintosh. Vous ne devez pas l'utiliser si vous désirez porter vos programmes sur d'autres machines.

Un signe **X** *dans la marge indique une primitive ou une caractéristique spécifique à l'environnement XWindows ou SunView.*

[1] Il est possible d'utiliser la forme [<X,Y> | <X',Y'>] pour représenter un rectangle (ce terme est une paire pointée de doublets représentant deux des points du rectangle). L'emploi de cette forme obsolète n'est pas conseillée.

Les descriptions des différents environnements ont été regroupées dans un seul document pour mettre en évidence les différences qui existent encore entre ces environnements.

2. Primitives de gestion des fenêtres

- **file_window(s1)**
- **file_window(s1,s2)**

  Crée, si elle n'existe pas déjà, une fenêtre de nom *s2* initialisée avec le texte contenu dans le fichier de nom *s1*. ***file_window(s1)*** équivaut à ***file_window(s1,s1)***.

- **front_window(s)**

  Unifie *s* avec le nom de la fenêtre se trouvant au premier plan.

get_screen(x,y)
get_screen(x,y,N)

Unifie *x* et *y* respectivement avec la largeur et la hauteur de l'écran en pixels. Dans la forme à trois arguments, *N* est unifié avec **1** si l'écran ne possède que deux couleurs (noir et blanc), un nombre supérieur à **1** sinon.

get_window(s,b,x1,y1,x2,y2)
get_window(s,b)

Unifie successivement (par backtracking) les paramètres *s,b,x1,y1,x2,y2* avec les valeurs correspondantes à chacune des fenêtres existantes. Voir ***set_window*** pour la signification des paramètres.

new_window(s,t)
new_window(s,t,v,x1,y1,x2,y2)
new_window(s,t,v,r)

Crée une nouvelle fenêtre, et une nouvelle unité d'entrée/sortie de nom *s* et de type *t*, où *s* est une chaîne différente de celles des unités existant déjà. Le nom *s* est inséré dans le menu Fenêtres.

© PrologIA

t est de la forme *type*, ou bien [*type*|*liste_attributs*].
type détermine le type de la fenêtre, et donc les opérations pouvant lui être appliquées.
Ses valeurs possibles sont :

- ***"EDIT"***
  création d'une fenêtre d'édition. On peut dans cette fenêtre écrire, lire, et éditer (c'est à dire agir directement sur la fenêtre, hors primitives d'E/S Prolog) : ces fenêtres fonctionnent comme des fichiers. L'écriture se fait toujours à la fin du texte contenu dans la fenêtre.

  "GRAPHICS"
  création d'une fenêtre avec les mêmes fonctionnalités que la fenêtre "*graphic*". Le but ***gr_window(s)*** permet de la définir comme unité graphique courante.
- ***"TTY"***
  création d'une fenêtre avec les mêmes fonctionnalités que la fenêtre "*console*".

  "MODAL"
  création d'une fenêtre graphique modale. Cette fenêtre reste au dessus de toutes les autres fenêtres, et tous les événements extérieurs à la fenêtre sont désactivés (excepté l'interruption utilisateur). Les événements extérieurs à la fenêtre sont réactivés lorsque la fenêtre est tuée. Les fenêtres modales peuvent être empilées.

Les attributs possibles dépendent du type texte (*"EDIT"* ou *"TTY"*) ou graphique (*"GRAPHICS"* ou *"MODAL"*) de la fenêtre :

- **X** ***<"DISTANCE",n>***
  n est un entier qui indique en pourcentage par rapport à la hauteur de la partie dessinable de la fenêtre, l'en-tête réservé pour une zone de création de boutons. Cet attribut est valable pour tout type de fenêtre.
- **X** ***<"NLINES",n>***
  n est un entier qui indique le nombre de lignes à sauver lors d'un scrolling dans une fenêtre de type *"TTY"*.

<"SHAPE",n>

n est un entier qui détermine la forme de la fenêtre. Les formes standard sont ***0*** : fenêtre avec titre et barres de défilement; ***1*** : fenêtre cadre. Par défaut les fenêtres modales ont l'attribut 1, et les fenêtres de texte l'attribut 0. Cet attribut est valable pour tout type de fenêtre. On notera que les fenêtres graphiques n'ont pas de barres de défilement.

- ***<"SAVE", r >***

  Cet attribut ne peut être appliqué qu'à une fenêtre graphique ou modale. Il définit le rectangle *r* de la fenêtre auquel est associé un bitmap de rafraîchissement (dans le cas des machines couleur une grande zone peut demander une place mémoire importante).

<"SAVE", b >

b est un entier (**0** ou **1**) indiquant si un bitmap de rafraîchissement est associé à la fenêtre. La valeur par défaut est $b = 0$.

- Si $b = 1$, un bitmap de rafraîchissement de la taille de la fenêtre est créé.

X Si $b = 1$, un bitmap de rafraîchissement de la taille de l'écran est créé.

- ***<"FONTSIZE", n >***

  Définit la taille de la police de caractères utilisée dans une fenêtre de texte.

- ***<"FONT", n >***

  Définit le numéro *n* de la police de caractères utilisée pour le texte d'une fenêtre type texte. La fonte associée à *n* est dépendante du système hôte.

v est un entier (***1*** ou ***0***) indiquant si la fenêtre doit être visible (***1***) ou pas (***0***).

Les quatres derniers paramètres que l'on peut remplacer par l'arbre *r* , sont :

x1,y1

qui représentent les coordonnées du coin supérieur gauche de la partie dessinable de la nouvelle fenêtre par rapport à l'écran.

© PrologIA

x2,y2

qui représentent les coordonnées du coin inférieur droit de la partie dessinable de la nouvelle fenêtre, l'origine étant située au coin supérieur gauche de l'écran.

Exemple de création d'une fenêtre graphique avec sauvegarde automatique:

```
> new_window("query",["GRAPHICS",<"SAVE",1>]);
```

Limitations : On ne peut ouvrir simultanément plus de 28 fenêtres, sachant que, dans l'environnement disponible sous XWindows ou SunView, un menu compte pour une fenêtre et une icône compte double.

- **save_window(s1, s2)**
- **save_window(s1)**

Sauve le contenu de la fenêtre de texte *s1* dans le fichier de nom *s2*. La forme à un argument est équivalente à ***save_window(s1,s1)***.

set_window(s,b,x1,y1,x2,y2)

Permet de manipuler la visibilité et l'emplacement des fenêtres déjà créées par Prolog III.

s

Nom de la fenêtre sous forme d'une chaîne de caractères (*"console"*, *"graphic"* pour les fenêtres prédéfinies).

b

Entier qui indique si la fenêtre sera visible (**0** si invisible, **1** si visible).

x1,y1

sont les coordonnées, par rapport à l'écran, du coin supérieur gauche de la zone dessinable de la fenêtre.

x2,y2

sont les coordonnées du coin inférieur droit de la zone dessinable de la nouvelle fenêtre, l'origine étant située au coin supérieur gauche de l'écran.

Exemple :

```
> set_window("graphic",1,50,50,300,300);
```

set_window(s,b)

Même chose que ci-dessus, mais pour manipuler les attributs d'une fenêtre. *b* peut être :

soit un entier : il indique alors si la fenêtre sera visible (**0** si invisible, **1** si visible),

soit un attribut (voir ***new_window***). Les attributs possibles sont respectivement,

pour les fenêtres graphiques et modales :

<"SAVE", 0>

<"SAVE", 1>

- ***<"SAVE", r>*** où *r* est un rectangle.

pour une fenêtre texte :

- ***<"FONT", n>***
- ***<"FONTSIZE", n>***

Exemple :

```
> set_window("console",<"FONTSIZE",12>);
```

X **create_window(s,t,v,r1,r2)**

X **create_window(s,t,v,r1)**

X **create_window(s,t,r1)**

Similaire à ***new_window***, mais permet de définir un système de coordonnées avec échelle.

© PrologIA

r1

est de la forme <*x1,y1,x2,y2*> où *x1,y1,x2,y2* sont des réels indiquant les coordonnées de la partie "*dessinable*" de la fenêtre dans un écran [0,1] x [0,1]. Le coin supérieur gauche de la fenêtre doit être dans l'écran, donc *x1* et *y1* dans l'intervalle [0,1] .

r2

est de la forme <*x1,y1,x2,y2*> où *x1,y1,x2,y2* sont des entiers (de type *court*, c'est-à-dire inférieur à 32700) définissant un système de coordonnées locales à l'utilisateur. Ce paramètre n'est à préciser que pour des fenêtres de type graphique (*"GRAPHICS"* ou *"MODAL"*). Si le paramètre *r2* n'est pas spécifié, les coordonnées graphiques dans la fenêtre seront données en pixels.

Exemple de création d'une fenêtre :

```
> create_window("wind","GRAPHICS",1,
          <0.1,0.1,0.6,0.7>,<0,0,100,100>);
```

quelle que soit la taille de l'écran, un rectangle dessiné par ***gr_rect(1,0,0,100,100)*** la remplira, ***gr_moveto(50,50)*** positionnera au centre etc...

- **reset_window(s)**

  Repositionne le pointeur de lecture au début d'une fenêtre d'édition de nom *s*. Echoue si *s* n'est pas une fenêtre d'édition créée par ***new_window***.

clear_window(s)

Efface le contenu de la fenêtre de nom *s*.

gr_window(s)

La fenêtre graphique *s* devient l'unité graphique courante.

gr_window_is(s)

Unifie *s* avec le nom de l'unité graphique courante. Au lancement de Prolog III, l'unité graphique courante est la fenêtre prédéfinie *"graphic"* .

kill_window(s)

Tue la fenêtre de nom *s*. Si la fenêtre *s* était l'unité graphique courante, alors la fenêtre prédéfinie *"graphic"* redevient l'unité graphique courante.
Le nom *s* ne peut être celui d'une fenêtre prédéfinie (i.e. ce doit être une fenêtre créée par ***new_window***.)

- **print_window(s,f,t)**
- **print_window(s)**

Imprime le contenu de la fenêtre de texte *s* avec la fonte *f* de taille *t*. La forme avec un seul argument imprime la fenêtre avec la fonte courante de la fenêtre.

- ***option fermeture***

Lorsque l'on clique dans la case de fermeture d'une fenêtre, l'unité correspondante est fermée et son nom est retiré du menu contenant les noms des fenêtres, excepté pour les fenêtres prédéfinies. Il est possible de rendre une fenêtre invisible sans la fermer en appuyant sur la touche *option* en même temps que la case de fermeture est cliquée.

Rafraîchissement de la fenêtre graphique

Les fenêtres graphiques peuvent être rafraîchies automatiquement par un *bitmap* auxiliaire en créant ces fenêtres avec l'attribut *<"SAVE",1>* (voir la primitive ***new_window***).
Il existe une coroutine attachée aux fenêtres graphiques qui est toujours déclenchée dès que ces fenêtres nécessitent un rafraîchissement. Cette coroutine est :

```
:gr_update(u)
```

où *u* représente le nom de la fenêtre à rafraîchir.

© PrologIA

La règle ***:gr_update*** n'est pas définie dans l'environnement standard. L'utilisateur peut donc, en écrivant une règle *<:gr_update, 1>*, définir par programme ce que doit être le rafraîchissement d'une fenêtre graphique *u* sans utiliser de bitmap auxiliaire.

3. Dessin et positionnement

Dans toutes les règles exposées ci-dessous, *x* et *y* représentent les coordonnées horizontales (*x*) et verticales (*y*). L'axe positif des *y* est dirigé *vers le bas*. Se rappeler que l'origine par défaut (*x*=0, *y*=0) coïncide avec le coin intérieur en haut et à gauche de l'élément de référence. Ce dernier est toujours la fenêtre graphique courante, sauf pour les primitives de positionnement des fenêtres pour lesquelles l'élément de référence est l'écran.

Les coordonnées *x* ou *y* peuvent être exprimées indifféremment par des nombres entiers, réels ou rationnels.

gr_moveto(x,y)

Positionne le point d'écriture (crayon) de la fenêtre graphique courante en *x,y*.

gr_lineto(x,y)

Tire un trait depuis la position courante jusqu'au point *x,y*, en déplaçant le crayon.

gr_move(x,y)

Modifie, sans tracer, la position du crayon de *x* pixels dans le sens horizontal et de *y* pixels dans le sens vertical. Si la position courante était (*x0*, *y0*) la nouvelle position est (*x0+x*, *y0+y*)

gr_line(x,y)

Tire un trait depuis la position courante (*x0*, *y0*) jusqu'au point (*x0+x*, *y0+y*) qui devient la nouvelle position.

gr_penloc(x,y)

Donne en coordonnées entières la position du crayon dans la fenêtre graphique courante.

gr_erase

Efface le contenu de la fenêtre graphique courante et repositionne le crayon en haut et à gauche.

gr_setorigin(x,y)

Change l'origine des axes de sorte que *x et y* représentent les coordonnées de l'angle haut et gauche de la fenêtre. L'aspect de la fenêtre n'est pas modifié.

gr_rect(n,r)

gr_rect(n,x1,y1,x2,y2)

Permet d'appeler les routines de bases pour tout dessin s'inscrivant à l'intérieur d'un cadre rectangulaire. *r* est un arbre correspondant à la position du rectangle. Cet arbre doit être de la forme :

<x1,y1,x2,y2>

x1,y1 sont les coordonnées du coin supérieur gauche, et *x2,y2* celles du coin inférieur droit.

Le premier argument *n*, qui représente un entier ou un identificateur, détermine la procédure appelée :

0, :frameRect, :frame

Dessine le périmètre du rectangle.

1, :paintRect, :paint

Remplit le rectangle avec le motif associé au crayon.

2, :eraseRect, :erase

Efface le rectangle et son contenu.

3, :invertRect, :invert

Inverse la couleur de chaque points de l'intérieur du rectangle. Si le rectangle a un coloriage autre que noir et blanc le résultat est dépendant de la machine .

4, :frameOval

Dessine le périmètre de l'ovale inscrit dans le rectangle.

5, :paintOval

Remplit l'ovale avec le motif associé au crayon.

© PrologIA

6, :eraseOval

Efface l'ovale et son contenu.

7, :invertOval

Inverse les bits de l'ovale.

8, :frameRoundRect

Identique à *:frameRect* mais avec des coins arrondis.

9, :paintRoundRect

Identique à *paintRect* mais avec des coins arrondis.

10, :eraseRoundRect

Identique à *eraseRect* mais avec des coins arrondis.

11, :invertRoundRect

Identique à *invertRect* mais avec des coins arrondis.

- ***12, :clipRect***

  Définit un rectangle dont le contour n'est pas tracé. Seule la partie des dessins ultérieurs située à l'intérieur du rectangle sera visible. Autrement dit, on définit un rectangle de visibilité pour tous les dessins à venir. Pour en annuler l'effet , refaire ***gr_rect(12,r)*** en donnant pour *r* un rectangle arbitrairement grand, comme par exemple *<-1000, -1000, 1000, 1000>*.

gr_polygon(n,L)

Dessin d'un polygone dans la fenêtre graphique courante. *L* est une liste binaire de doublets *<x,y>* indiquant les coordonnées des points définissant le polygone.

Le premier argument *n*, qui représente un entier ou un identificateur, détermine la procédure appelée :

0, :frame

Dessine le périmètre du polygone.

1, :paint

Remplit le polygone avec le motif associé au crayon.

2, :erase

Efface le contenu du polygone (certaines lignes brisées formant le contour du polygone peuvent subsister).

3, :invert

Inverse la couleur de chaque point de l'intérieur du polygone. Le résultat est dépendant de la machine si le polygone a un coloriage autre que noir et blanc.

gr_arc(n,r,a1,a2)[1]

gr_arc(n,x1,y1,x2,y2,a1,a2)[2]

Dessine un arc d'ellipse inscrit dans le rectangle *r* entre les angles *a1* et *a2*. Les angles *a1* et *a2* sont donnés en degrés.

Le premier argument *n*, qui représente un entier ou un identificateur, détermine la procédure appelée :

0, :frame

Dessine l'arc d'ellipse.

1, :paint

Remplit le secteur défini par l'arc d'ellipse avec le motif associé au crayon.

2, :erase

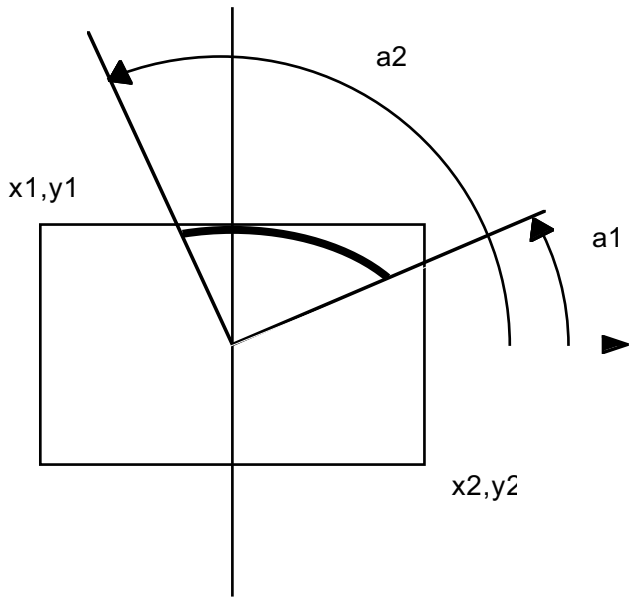
Efface l'arc d'ellipse et le contenu du secteur.

3, :invert

Inverse les bits du secteur défini par l'arc d'ellipse. Le résultat est dépendant de la machine si le secteur a un coloriage autre que noir et blanc.

Exemple :

```
> gr_arc(:frame,100,100,200,170,20,110);
```



1 2 Cette primitive n'est pas encore disponible sous Sunview.

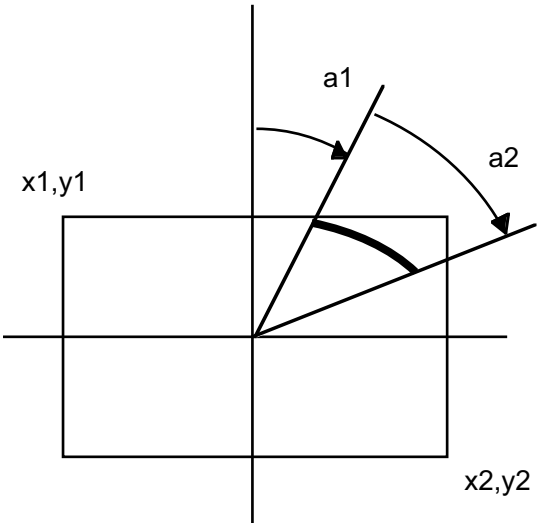
© PrologIA

- **gr_arc'(n,r,a1,a2)**
- **gr_arc'(n,*x1*,y1,*x2*,y2,a1,a2)**

  Dessine un arc d'ellipse inscrit dans le rectangle *r* , depuis l'angle *a1* et d'une amplitude *a2*. Les angles *a1* et *a2* sont donnés en degrés. Cette primitive se différencie de ***gr_arc*** par le fait qu'elle correspond exactement à la définition de la *Toolbox* (origine sur l'axe vertical, *a2* relatif à la position *a1*, sens positif dans le sens des aiguilles d'une montre).

Exemple :

```
> gr_arc'(:frame,100,100,200,170,20,50);
```



- **gr_icon(n,*x1*,y1,*x2*,y2)**
- **gr_icon(n,r)**

  Dessine dans l'unité graphique courante, et inscrit dans le rectangle indiqué une icône décrite dans une ressource 'ICON' de numéro *n*. Cette ressource est recherchée d'abord dans l'exécutable Prolog III, puis dans le fichier système. Le système contient toujours les icônes :

  0 Panneau stop.
  1 Notification.
  2 Panneau danger.

  Le meilleur résultat est obtenu en donnant comme rectangle un carré de coté égal à 32.

4. Usage de la souris

Règles pour manipuler les fenêtres à l'aide du clavier et de la souris

X Les règles suivantes ne concernent que l'utilisateur *XWindows ou SunView* disposant d'une souris à 3 boutons.

Déplacement	bouton central maintenu appuyé
Agrandissement	touches SHIFT + CONTROL + bouton central maintenu appuyé
	ou bouton gauche dans le coin inférieur droit de la fenêtre
Menus	clic du bouton droit pour les faire apparaître
	clic du bouton gauche pour valider le choix
Ramener une fenêtre au premier plan	
	touche SHIFT + clic du bouton central
Faire passer une fenêtre en arrière-plan	
	touche CONTROL + clic du bouton central

Repérage de la souris

Dans ce qui suit, les variables commençant par *r* représenteront des arbres correspondant à des rectangles, c'est à dire de la forme *<x1,y1,x2,y2>* .

© PrologIA

gr_click(b,x,y)

Teste si un bouton de la souris a été pressé depuis le dernier appel de ***gr_erase*** ou ***gr_click***. Si c'est le cas, *x* et *y* sont unifiés avec les coordonnées entières du point où se trouvait la souris lorsque cet événement, appelé "*clic*", s'est produit. Sinon agit en fonction de *b*, entier qui doit être connu au moment de l'appel. La signification des valeurs de *b* est la suivante :

b = ***1***

Attendre qu'un nouveau clic se produise dans la fenêtre graphique courante.

b = ***0***

Ne pas attendre et réaliser un backtracking s'il n'y a aucun clic en attente.

gr_click(b,x,y,m)

Identique à ***gr_click(b,x,y)*** mais donne en plus l'état des principales touches de modification en unifiant *m* avec la somme des valeurs suivantes :

1

La touche SHIFT est appuyée.

2

La touche CONTROLE[1] est appuyée.

- ***4***

  La touche OPTION est appuyée.

- ***8***

  La touche COMMANDE est appuyée.

- ***16***

  La touche MAJUSCULE-VERROUILLEE est enfoncée.

Si *m* = ***0*** aucune touche de modification n'est appuyée durant le clic.

1 Cette touche est absente sur le clavier du Macintosh +.

gr_getmouse(x,y)
gr_getmouse(x,y,b)

Donne la position actuelle de la souris dans la fenêtre graphique courante en coordonnées entières relatives à l'origine de la fenêtre. Rend pour *b* la valeur **0** si le(s) bouton(s) de la souris sont relevés au moment de l'appel, sinon le numéro du bouton pressé en commençant par la gauche pour une souris à plusieurs boutons (le premier bouton a le numéro **1**).

5. Modes de dessin et d'écriture

- **gr_color(d,c)**

  Détermine la couleur du fond (si *d*= **0**) ou du crayon (si *d*=**1**) de la fenêtre graphique courante. L'argument *c* est un entier représentant une des 8 couleurs possibles parmi les valeurs :
  33 : noir, **30** : blanc, **69** : jaune, **137** : magenta, **205** : rouge, **273** : cyan, **341** : vert, **409** : bleu.

X **gr_color2(f,c)**

  Impose la couleur *f* du fond et *c* du crayon pour la fenêtre graphique courante. Les arguments *f* et *c* sont des entiers représentant une des couleurs possibles parmi les valeurs :
  0 : blanc, **1** : rouge, **2** : vert, **3** : bleu, **4** : rose, **5** : orange, **6** : marron, **7** : rose1, **8** : violet, **9** : vert1, **10** : jaune, **11** : bleu1, **12** : jaune1, **13** : magenta, **14** : cyan, **15** : noir.

- **gr_color3(r,v,b)**

  Définit la couleur du crayon graphique en composantes *r* rouge, *v* vert, *b* bleu. Si les valeurs sont des nombres flottants, elles doivent être comprises entre 0.0 et 1.0, et correspondent à des pourcentages d'*intensité* pour chaque composante (0.0 est l'absence de cette composante, 1.0 est sa présence avec une intensité maximale). Si les valeurs sont entières, ce sont les valeurs spécifiques de la machine qui doivent être utilisées (de 0 à 65535 sur Macintosh).

© PrologIA

- **gr_choosecolor(p, c, i, f)**

  Affiche le dialogue de sélection de couleur au point *p* (de la forme <*x*,*y*>) de l'écran. *c* est une chaîne qui sera affichée dans le dialogue. *i* et *f* sont des triplets d'entiers compris entre 0 et 65535; un triplet <*r*, *v*, *b*> représente une couleur en composantes *r* rouge, *v* vert, *b* bleu. *i* est la couleur initiale du dialogue et doit être connu au moment de l'appel, *f* sera unifié avec la couleur choisie. Echoue si l'utilisateur clique le bouton *Annuler*.

gr_pen(l,m)

- **gr_pen(l,h,m)**

  Définit la taille en pixels du crayon graphique, et son motif.

  l = largeur[1]

  h = hauteur

  m = motif.

  0 blanc

  1 noir

  2 gris

  3 gris clair

  4 gris foncé

gr_stringwidth(s,n)

Unifie *n* avec la longueur de la chaîne *s* en fonction du jeu de caractère , de la taille et du style définis pour la fenêtre graphique courante.

gr_mode(b,m)

Définit le mode d'application du motif sur le fond, pour l'unité graphique courante.

b = ***0***

Concerne le dessin et donc le motif du crayon.

b = ***1***

Concerne le texte et donc le motif correspondant au caractère.

---

1 La largeur n'a pas d'effet sous Sunview.

Il existe quatre opérations de base : Copy, Or, Xor et Bic. L'opération Copy remplace simplement les pixels de destination par les pixels du motif ou de la source, dessinant sur la destination sans se préoccuper de son état initial. Les opérations Or, Xor, et Bic laissent inchangés les pixels de destination situés sous la partie blanche du motif ou de la source. Elles traitent différemment les pixels situés sous la partie noire :

Or (ajout)	les remplace par des pixels noirs.
Xor (inversion)	les remplace par leur contraire.
Bic (effacement)	les remplace par des pixels blancs.

Suivant le mode, l'opération est réalisée à partir du motif lui-même (src), ou bien à partie de son inverse (notSrc). Voici les valeurs de *m* pour les différents modes.

m	modes de transfert
0	srcCopy
1	srcOr
2	srcXor
3	srcBic
4	notSrcCopy
5	notSrcOr
6	notSrcXor
7	notSrcBic

- **gr_text(f,t,s)**

  Permet de définir la fonte, la taille et le style des sorties dans la fenêtre graphique courante.

  f = numéro de la fonte

  t = taille de la fonte

  s = liste de numéros de style.

© PrologIA

N°	fonte	N°	style
0	systemFont	0	gras
1	applFont	1	italique
2	newYork	2	souligné
3	geneva	3	relief
4	monaco	4	ombré
5	venice	5	condensé
6	london	6	étalé
7	athens		
8	san Franscico		

...

La liste binaire vide pour *s* représente le texte normal. La combinaison par défaut est ***gr_text(4,12,[])***.

Par exemple ***gr_text(3,18,[1,3,4])*** correspond à *geneva* taille 18, italique, relief et ombré.

Il est possible d'appeler cette règle en donnant en argument des identificateurs auxquels on a assigné les constantes appropriées.

Exemple :

```
> assign(Geneva,3) assign(Italic,1)
assign(Relief,3);
{}
> gr_text(Geneva,18,[Italic,Relief])
  output("graphic") outm("Hello")
  output("console");
```

affiche *Hello* dans la fenêtre graphique.

X gr_font(f)

Charge la fonte de nom *f* pour la fenêtre graphique courante. La fonte de base est "`courier`"; les fontes proposées sont des variations sur la hauteur et l'épaisseur des caractères :

"ft7r", "ft12r", "ft12b", "ft14r", "ft14b", "ft19r".

7, 12, 14, et 19 désigne la taille de la fonte, r représente une épaisseur standard et b une plus grande épaisseur.

Exemple :

```
> gr_font("ft12r");
```

6. Primitives spéciales de lecture et d'écriture au clavier et à la souris

Simulation de boutons

gr_draw_buttons(b)

Dessine un ou plusieurs boutons dans la fenêtre graphique courante.

La définition d'un bouton est donnée par une règle Prolog III de nom *b* et d'arité 3.

Les 3 arguments sont, respectivement :

- le rectangle dans lequel il s'inscrit
- la chaîne affichée par le bouton
- un terme permettant de coder une information supplémentaire (par exemple, une action à associer à ce bouton).

Tous les boutons définis par le paquet *<b,3>* sont dessinés.

Exemple de base de données de boutons :

```
wtButton(<5,160,50,175>, "Edit", edit) -> ;

wtButton(<5,180,50,195>, "Set", []) -> ;
```


© PrologIA

Lorsque cette base est chargée, les deux boutons sont dessinés par le but :

```
> gr_draw_button(wtButton)  ;
```

gr_button_hit(b,<x,y>,t)

b est l'identificateur d'accès à la base de données de boutons décrite ci-dessus.

Cette règle s'efface si *x* et *y* sont les coordonnées d'un point intérieur à l'un des boutons de la base de données *<b,3>*.

On unifie ensuite *t* avec le troisième argument du bouton concerné.

La primitive ***gr_button_hit*** combinée avec ***gr_click*** permet d'écrire facilement une boucle de scrutation des boutons.

Saisie de texte

gr_editf(<s1,p1,p2>,r,s,k)

gr_editf(<s1,p1,p2>,x1,y1,x2,y2,s,k)

Crée un rectangle d'édition dans la fenêtre graphique courante. La fin d'édition est provoquée par un retour chariot, un caractère de tabulation ou un clic en dehors du rectangle d'édition. Si *k* n'est pas une variable, le rectangle et son texte sont seulement dessinés sans édition.

s1

est la chaîne qui sert à initialiser le champ d'édition.

p1, p2

sont des entiers définissant la position de début et de fin de sélection du texte (représentée en vidéo inverse). La valeur **0** pour *p1* indique la position qui précède le premier caractère du texte; *p2* est la position du caractère qui suit le dernier dans la sélection. Initialiser avec un champ vide correspond aux valeurs: *<"",0,0>*.

r

définit la position du rectangle englobant. Le texte est cadré automatiquement en mode paragraphe avec justification à gauche.

s

est la variable qui sera unifiée avec la chaîne éditée.

k

Si *k* est une variable, celle-ci est unifiée en sortie avec le mode de terminaison :

- ***0*** si retour chariot
- ***1*** si clic souris dans la fenêtre[1] (en dehors du rectangle d'édition)
- ***2*** si caractère TAB.

Si *k* est une constante en entrée, sortie immédiate après affichage de la zone.

Une forme plus simple de la règle est également utilisable pour une chaîne initiale vide : ***gr_editf(<>,x1,y1,x2,y2,s,k)***

get_key(c)

get_key(a,t,m)

get_key sert à prendre un caractère au vol. Il n'y a aucun écho de ce caractère dans aucune fenêtre que ce soit. Cette primitive permet donc de saisir un mot-de-passe. Le fonctionnement détaillé est donné dans la description de ***stty*** .

get_key(c)

rend dans *c* un caractère.

get_key(a, t, m) rend trois entiers :

- dans *a* le code ascii étendu
- dans *t* le numéro de touche (*virtual key code* décrit dans Inside Macintosh vol. V p.192))
- dans *m* l'état des touches de modifications (voir la primitive ***gr_click***)

1 On pourra lire sa position avec ***gr_click(0,x,y)*** .

© PrologIA

- **stty("USEGETKEY", t)**

  réglemente d'usage du prédicat ***get_key***.

  $t = 0$

  mode par défaut, dans lequel ***get_key*** échoue toujours. Les caractères tapés sont insérés dans les fenêtres texte le cas échéant. Dans ce mode, l'utilisateur peut éditer des fenêtres texte pendant que Prolog III travaille.

  $t < 0$

  get_key prendra un caractère qui ne peut être plus âgé que t ticks (1/60sec) avant l'appel. Echoue si, au moment de l'appel, aucun caractère n'est disponible, ou si le caractère est trop "*vieux*" .

  $t > 0$

  get_key attendra un caractère pendant au plus t ticks. Echouera si aucun caractère n'est disponible au bout de ce laps de temps. ***get_key*** n'attend pas que le délai soit écoulé pour réussir, si on lui fournit un caractère dans les temps. Un caractère peut avoir été tapé à l'avance. Ce mode permet de saisir un mot-de-passe en un temps donné.

Exemples :

```
> stty("USEGETKEY",  -60);
```

Le prochain appel à ***get_key*** échouera si aucun caractère non lu n'a été tapé moins d'une seconde avant cet appel (1 s = 60 ticks).

```
> stty("USEGETKEY",  300);
```

Le prochain appel à ***get_key*** échouera si aucun caractère non lu ne peut être obtenu dans les cinq secondes suivant cet appel (1 s = 60 ticks). Le caractère peut avoir été tapé à l'avance. ***get_key*** réussira de suite après avoir obtenu le caractère sans attendre que le délai soit écoulé.

- **stty("FLUSH")**

  Enlève tous les caractères de la queue d'événements. Ils ne sont plus disponibles pour ***get_key***. (permet à ***get_key*** de saisir une réponse qui n'a pas été tapée à l'avance.)

- **gtty(s, v)**

  Récupère la valeur d'un paramètre concernant le terminal :

  gtty("USEGETKEY", v) rend 0 ou un nombre signé de ticks.

  gtty("TERM", x) rend actuellement *"TTY"* ou *"MPW"*.

Boîtes de choix

gr_list(r, Liste, InSel, InTop, OutSel, OutTop, op)

La règle ***gr_list*** gère un choix d'élément(s) dans une liste. Cette dernière s'affiche dans la fenêtre graphique courante.

r

Le rectangle englobant (ascenseurs compris) de la forme *<x1,y1,x2,y2>*.

Liste

La liste d'items. C'est une liste binaire de constantes éventuellement mixtes (chaîne, entier, réel, ident ...).

InSel

C'est la sélection initiale, sous la forme d'une liste binaire d'entiers qui sont les rangs (à partir de 1) dans *Liste* des éléments que l'on veut sélectionner. Mettre *[]* si aucun élément n'est sélectionné au départ.

InTop

C'est le rang de l'élément qui doit apparaître tout en haut de l'affichage de la boîte (*scrolling* initial). On peut mettre **1** par exemple.

OutSel

Donne en sortie la liste des rangs des items sélectionnés (même convention que pour *InSel*).

OutTop

Donne en sortie le rang du premier item visible tout en haut de la boîte de choix.

op

Décrit l'opération à réaliser. 4 possibilités :

© PrologIA

<0> Dessine seulement (pas de saisie possible).

<1,k> Dessine et prend la main jusqu'à ce qu'un événement de type entrée de caractère ou clic à l'extérieur de la liste se produise. **Ni le clic ni le caractère ne sont lus**. Rend pour *k* :

0 : retour chariot tapé
1 : clic en dehors de la liste
2 : TAB à été tapé
3 : un autre caractère que RC ou TAB à été tapé

<2,x,y,m> Dessine la liste et traite le clic *x,y,m* (donné par exemple par ***gr_click(b,x,y,m)***). Sort tout de suite après.

<3,x,y,m,k> Dessine, traite le clic *x,y,m* (donné par exemple par ***gr_click(b,x,y,m)***) et prend la main jusqu'à ce qu'un événement de type clavier ou un clic à l'extérieur du rectangle se produise. **Ni le clic ni le caractère ne sont lus**. Rend pour *k* :

0 : retour chariot tapé
1 : clic en dehors de la liste
2 : TAB à été tapé
3 : un autre caractère que RC ou TAB a été tapé.

- **sys:gr:gr_list'(r, L, f, Sel, ISel, ITop, OSel, OTop, op)**

  est la primitive de base utilisée. Son 3ème argument est actuellement inutilisé (il doit être libre) et son 4ème est le type de sélection des items. Si vous désirez modifier le type de sélection par défaut vous pouvez remplacer la règle ***gr_list*** de *"sys"* (à l'aide de ***reinsert***, ou de ***suppress*** puis ***assert***) par celle-ci, dans laquelle vous aurez changé le 28 par le type de votre choix :

```
gr_list(r, L, ISel, ITop, OSel, OTop, op) ->
        sys:gr:gr_list'(r, L, _, 28, ISel, ITop,
                        OSel, OTop, op);
```

  Quelques types de sélections possibles parmi d'autres (Inside Macintosh IV-267) :

  0 *shift* et *command*
  28 *shift* ou *command* seulement (valeur par défaut)

-128 un seul item a la fois peut être sélectionné.

- **gr_popupItem(r,l,n1)**
- **gr_popupItem(r,l,n1,n2)**

Crée un champ à valeurs proposées. Ceci correspond à ce qui est appelé un menu *"pop-up"*, dans la *Toolbox*. Il ne faut pas confondre ce type d'objet avec les menus fugitifs liés à des fenêtres, menus activés par *option-clic* et permettant de lancer des actions de façon asynchrone.

Descriptions des arguments :

l est une liste de constantes.

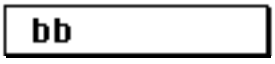
r est le rectangle dans lequel est affiché l'item numéro *n1* de la liste *l*.

Si *n2* est absent, l'item est simplement dessiné.

Si *n2* est présent, un menu présentant la liste des choix *l* vient recouvrir l'item, et l'inversion des champs est gérée tant que le bouton de la souris reste appuyé. Lorsque le bouton est relâché, le champ sélectionné est dessiné dans le rectangle à la place du précédent, et *n2* est unifié avec le numéro de l'élément sélectionné (*n1* si aucun champ n'est sélectionné).

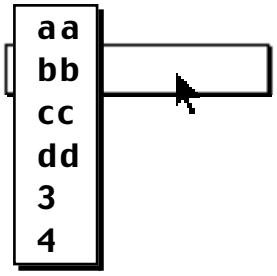
Exemple lorsque la fenêtre graphique est rendu visible :

```
> gr_popupItem(<20,20,120,40>,
        ["aa","bb","cc","dd",3,4],  2)
  gr_click(1,x,y)
  gr_popupItem(<20,20,120,40>,
        ["aa","bb","cc","dd",3,4],  2,  i);
```



Lorsque vous cliquez dans la fenêtre en maintenant le bouton appuyé :

© PrologIA



Impression de fenêtres

- **gr_print(s)**
- **gr_print(s,r_x,r_y,x0,y0)**
- **gr_print(s,r1,r2)**

  Imprime le contenu de la fenêtre graphique *s* .
  On peut spécifier, grâce à la troisième forme, la zone à imprimer, qui est le contenu du rectangle *r1*, le résultat sur papier se situant dans le rectangle *r2*. Les réductions et translations éventuelles sont automatiques. La forme avec un seul argument imprime la fenêtre sans modification de format.
  La forme à 5 arguments permet de spécifier la réduction selon les axes des X, des Y sous la forme de deux nombres *r_x* et *r_y* compris entre 0.0 et 1.0. On peut donner, en *x0* et *y0* la position de l'origine de l'impression sur le papier.

Exemple :

```
> gr_print("graphic", 0.5, 0.5, 0, 0);
```

imprimera la fenêtre *"graphic"*, réduite au quart (chaque dimension étant réduite de moitié).

7. Primitive de description de menus

Un menu regroupe une suite d'items disposés verticalement. Un item peut être soit terminal, soit lui-même un menu (on parle alors de menu hiérarchique). A tout item terminal d'un menu hiérarchique est associé un but Prolog III qui est activé immédiatement, en suspendant le programme en cours, lorsque l'item est sélectionné.
Lorsque la barre de menu existe, chaque item de cette barre se comporte comme un item d'un menu hiérarchique.

Il est possible d'attacher un menu fugitif (*pop-up*) particulier à toute unité de type fenêtre. La primitive ***set_menu*** décrite ici, permet de définir ou de redéfinir de tels menus .

L'activation d'un menu fugitif se fait soit en appuyant sur le bouton droit de la souris (à l'intérieur de la fenêtre concernée), lorsque la souris comporte plusieurs boutons, soit en appuyant simultanément une touche *option* et le bouton de la souris, si la souris ne possède qu'un seul bouton.

set_menu(u,c,v)

Permet de définir ou de redéfinir par *v* la valeur de l'item indiqué par *c* dans le menu de l'unité *u*.

u (unité)

Représente l'unite concernée : l'entier **0** pour la barre de menu (et ses descendants), le nom d'une fenêtre sinon.

c (chemin)

Liste binaire de chaînes indiquant un chemin de sélection dans le menu hiérarchique de l'unité *u*. L'élément terminal de ce chemin est remplacé par *v*.
Si une telle hiérarchie n'existe pas, elle est créée.
Si la liste est *[]*, c'est tout le menu qui est concerné.

v (valeur)

Arbre de description de la nouvelle valeur. Celle-ci peut être soit une feuille, soit une hiérarchie.

Valeur de type feuille.

Une feuille est représentée par un tuple de la forme :

<"Nom item", Identificateur, n>

La chaîne *"Nom item"* donne le nom de l'item.

© PrologIA

Identificateur est le nom d'une règle Prolog d'arité 1 attachée à l'item.
n est un entier (0 ou 1) qui indique si l'item est ou non coché (une marque précède le nom de l'item coché).
Lorsqu'un item d'un menu est sélectionné et validé, le programme en cours est suspendu, et le but ***Identificateur(u)*** est immédiatement activé. Si le but se termine sans erreur, le programme reprend normalement.
Il est possible d'utiliser des formes simplifiées :
soit avec un tuple à deux arguments :
<"Nom item", Identificateur>
(équivalent à : *<"Nom item", Identificateur, 0>*),
soit avec simplement un identificateur : *Identificateur*
(équivalent à : *<"Identificateur", Identificateur, 0>*).

Exemple, en supposant que l'on ait la fenêtre d'édition "menu.p3" ouverte :

```
> set_menu(0,
      ["Controle", "Inserer la fenetre"],
      <"Inserer la fenetre", insertion_fen>)
  set_menu("menu.p3", ["Inserer"],
      <"Inserer", insertion_fen>);
{}
insertion_fen(0) ->
  front_window(u)
  reinsert(u);
insertion_fen(u) ->
  string(u)
  reinsert(u);
```

Valeur de type hiérarchie

Une hiérarchie est représentée par un tuple à deux arguments de la forme :
<"Nom item", Liste_de_valeurs>

Chaque élément[1] de *Liste de valeurs* représente lui-même une valeur de type feuille ou une valeur de type hiérarchie.

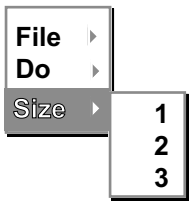
Il est possible d'utiliser une forme simplifiée :

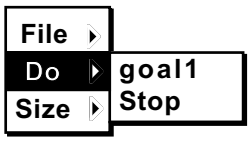
Liste_de_valeurs

Dans ce cas le nom de l'item est celui indiqué à la fin du chemin *c*. Ceci revient à apposer la hiérarchie sous l'item indiqué.

Exemple :

```
>  set_menu("graphic", [],          [
      <"File",    [reinsert]>,
      <"Do",      [goal1,<"Stop",goal2>]>,
      <"Size",    [   <"1",size1>,
                      <"2",size2,1>,
                      <"3",size3>  ]>  ] )
insert;
size1(u)  ->  gr_window(u)  gr_text(1,10,[]);
size2(u)  ->  gr_window(u)  gr_text(1,12,[]);
goal1(u)  ->
...
```





Dans les primitives ci-dessous, le premier et le deuxième argument suivent les mêmes conventions que pour ***set_menu*** :

---

1 Sous Sunview, chaque élement de la liste de valeurs doit être une feuille (un seul niveau).

© PrologIA

check_item(u,c,v)

Cette primitive positionne ou enlève le "*cochage*" d'un item.

v indique la présence du cochage d'un item : **0** indique non coché, **1** indique coché avec la marque standard de la machine.

- Si *v* est un entier plus grand que 1, le caractère ayant le code *v* est utilisé comme marque.

Exemple :

```
> check_item(0,["Fenetres","trace"],1);
```

- **command_menu(u,c,v)**

v est le code du caractère utilisé en conjonction de la touche commande (pomme), utilisée comme raccourci pour activer l'item spécifié par les deux premiers arguments.

Exemple de création du raccourci T pour activer la trace Prolog III :

```
> char_code(`T`,n)
command_menu(0,["Controle","trace"],n);
```

enable_menu(u,c,v)

Permet de désactiver (*v =**0***) ou de réactiver (*v =**1***) un menu ou un item.

Exemple pour désactiver tout le menu d'édition :

```
> enable_menu(0,["Editer"],0);
```

- **style_menu(u,c,v)**

  Spécifie le style d'un item de menu. *v* est un entier dont chaque bit représente un des styles spécifiés avec l'ordre défini dans la primitive ***gr_text***.

  Exemple mettant en italique souligné l'item *trace* du menu *Fenêtres* (bit 1 + bit 2 positionnés) :

```
> style_menu(0,["Fenetres","trace"],6);
```

- **clear_menubar**

  Appelle ***save_menubar*** puis efface la barre de menu standard. Cette primitive permet la redéfinition complète de la barre de menus.

- **save_menubar**

  Mémorise la barre de menu courante. Cette règle ne peut être appelée qu'une fois.

- **restore_menubar**

  Restaure la barre de menu sauvée avec ***save_menubar***.

- **add_stdmenu(s)**

  Cette primitive permet de réinsérer un ou plusieurs menus standard après un appel à ***clear_menubar***. La chaîne *s* doit être l'une des suivantes :

"apple"	: pour réinsérer le menu pomme
"file"	: pour réinsérer le menu Fichier
"edit"	: pour réinsérer le menu Editer
"find"	: pour réinsérer le menu Recherche
"control"	: pour réinsérer le menu Contrôle
"window"	: pour réinsérer le menu Fenêtres

© PrologIA

8. Gestion des boîtes de dialogue

Définition d'un dialogue standard

gr_dialog(d,l1,l2)
gr_dialog(p,d,l1,l2)

L'utilisation de cette primitive nécessite le chargement préalable du module *dial.mo3*. Ce module contient le gestionnaire de dialogue Prolog III permettant de créer et activer les dialogues décrits par le deuxième argument *d*.

Dans sa deuxième forme la primitive ***gr_dialog*** accepte un argument supplémentaire *p* permettant de paramétrer la position de l'angle supérieur gauche du dialogue sous la forme d'un doublet *<x,y>*.

d

est soit un arbre décrivant un dialogue, soit l'identificateur d'une règle unaire avec un tel arbre en argument.

l1

est une liste (éventuellement vide) de paires

[nom_de_zone | valeur_initiale]

où *nom_de_zone* représente un identificateur ou une constante. Cette liste est utilisée pour définir ou redéfinir le contenu des zones.

l2

En sortie, si *l2* est une variable, elle est unifiée avec la liste des paires *[nom_de_zone | valeur_finale]* du dialogue. Si *l2* est une liste de paires *[nom_de_zone | X]*, chaque paire est unifiée avec le couple correspondant.

Un dialogue est décrit avec des objets primitifs (zone de texte, zone d'édition, boutons, ...) et des primitives d'agencement en colonne ou en rangée. Ces primitives réalisent un ajustement dynamique de la taille des zones en fonction de leur contenu. Les zones dont la valeur peut être modifiée sont identifiées par un nom, ce qui permet de redéfinir leur valeur au moment de l'appel.

Exemple :

```
> reload("dial.mo3");
{}
> gr_dialog(["Hello"|buttonD("OK")],[],L);
```

```
> gr_dialog(<50,50>,
  ["Name:", editf(10,field1), buttonD("OK")],
  [[field1|"John"]],
  L);
```

Les primitives de description d'objet sont les suivantes :

:text(s) ou **s**

Décrit une ligne de texte non éditable représentée par *s* où *s* est une chaîne.

:text(i)

Décrit une ligne de texte non éditable dont le nom de zone est *i* où *i* est un identificateur. Si une valeur est associée à *i* dans *l1* c'est cette valeur qui est la valeur initiale du champ, sinon c'est la chaîne vide. Il n'est jamais associé de valeur finale à cette zone, puisqu'elle n'est pas modifiable.

© PrologIA

:cb(i), **:cb(i,v)**

Décrit une *boîte à cocher* représentée par une des deux valeurs entières possibles de *v* qui sont soit **1** soit **0**. Si la valeur de *v* n'est fournie ni en argument, ni dans la liste *l1*, c'est la valeur 0 qui est donnée par défaut. Lorsque *v* est redéfinie dans *l1*, c'est toujours cette dernière valeur qui est prise en compte.

:rb(<g,i>), **:rb(<g,i>,v)**

Décrit un "*radio-bouton*" *i* d'un groupe *g* (où *g* est un identificateur, et *i* une constante). Ce radio-bouton de valeur **0** ou **1**, fait partie du groupe *g*, avec au plus une valeur possible à 1 dans le groupe. Si la valeur de *v* n'est fournie ni en argument, ni dans la liste *l1*, c'est la valeur 0 qui est prise par défaut. Lorsque *v* est redéfinie dans *l1*, c'est toujours cette dernière valeur qui est prise en compte. En sortie, *nom_de_zone* est représenté par <*g*,*i*>[1], et *valeur_finale* par 0 ou 1, et ceci pour chacun des radio-boutons du groupe.

:rb1(<g,i>), **:rb(<g,i>,v)**

Même chose que *rb* mais avec deux différences : les éléments du groupe ne peuvent être tous à zéro; seul l'élément ayant la valeur 1 figure dans la liste de sortie. *nom_de_zone* est alors représenté par *g*, et *valeur_finale* par *i*. Ce format de sortie est donc différent et plus simple que celui fourni par *rb*.

:editf(n,i), **:editf(n,i,s)**

Décrit une zone d'une ligne de texte éditable de nom *i*, où *i* est un identificateur. *s* représente la valeur initiale de la zone d'édition. Cette valeur initiale est la chaîne vide si elle n'est spécifiée ni en argument, ni dans la liste *l1*. L'entier *n* représente le nombre minimum de caractères de la zone. Si *s* est plus grand, la zone est dynamiquement étendue.

---

1 Contrairement à Prolog II+, le doublet <g,i> ne représente pas le même arbre que g(i).

:button(s,i)

Décrit un bouton avec le label *s* auquel est associé l'action définie par l'identificateur *i*. Lorsque le bouton est activé, une action est réalisée :

- Si *i* est ***fail***, le dialogue est terminé, et ***gr_dialog*** échoue.
- Si *i* est *[]*, rien n'est fait.
- Sinon, le but ***i(l)*** est activé, *l* étant la liste binaires des paires *[nom|valeur]* des zones du dialogue au moment où le bouton est cliqué.

Lorsque le but correspondant à la règle *<i, 1>* a été exécuté, la gestion du dialogue reprend en l'état. Pour arrêter le dialogue en cours, on peut programmer dans *i* l'exécution d'un ***block_exit(<fail,L>)*** qui provoquera un bactracking de ***gr_dialog***, ou un ***block_exit(<[],L>)*** qui conduira à une terminaison normale.

:button2(s,i)

Cet item a le même comportement que l'item *:button*, mais cette fois-ci la règle *i* est appelée avec *deux* arguments. Le premier est une liste décrivant l'état des items du dialogue. Le deuxième est à l'appel une variable qui doit être instanciée par la règle *<i,2>* avec la liste des items dont la valeur est modifiée. Cette règle est écrite par l'utilisateur, et permet donc d'attacher une sémantique quelconque aux boutons. La règle définit une relation entre le premier argument (argument d'entrée représentant la liste des valeurs actuelles), et le deuxième argument (argument de sortie représentant la liste des valeurs nouvelles).

:buttonD, **:buttonD(s)**, **:buttonD(s,i)**

Décrit un bouton de terminaison de dialogue avec le label *s* auquel est associé l'action *i* définie comme pour ***:button(s,i)***. Usuellement *s* est la chaîne "OK", et *i* est *[]*. La frappe du retour chariot actionne ce bouton .

:glist(l, c, i, l_init)

Définit une liste de valeurs avec barre de défilement verticale. La signification des arguments est :

© PrologIA

l

Entier qui représente le nombre d'items qui doivent être visibles. La hauteur du rectangle est calculée automatiquement en fonction de ce paramètre.

c

Entier qui représentent le nombre de caractères qui doivent être visible pour un item. La largeur du rectangle est calculée automatiquement en fonction de ce paramètre.

i

Identificateur servant à donner un nom interne au gestionnaire. Il est utilisé pour donner le résultat en sortie.

l_Init

Liste de constantes (chaînes, identificateurs, nombres) qui représente la liste des valeurs initiales. Il est possible de proposer une sélection initiale en fournissant pour *l_Init* un triplet décrivant un état du gestionnaire. Ce triplet doit alors avoir la forme suivante :

<ListeInit, NoSelectionnés, NoHaut>

- Le premier argument est une liste de constantes représentant la liste des valeurs initiales.
- Le deuxième argument est la liste ordonnée des numéros d'ordre des items composant la sélection initiale. Cette liste peut comporter 0 (pas de sélection), 1, ou plusieurs éléments.
- Le troisième argument indique le numéro de l'item à présenter en haut du rectangle d'affichage.

Ce triplet correspond à la représentation interne de l'état du gestionnaire. Ainsi, la paire :

[Identificateur | <ListeInit, NoSelectionnés, NoHaut>]

intervient dans la liste des valeurs transmise aux actions associées aux boutons. La valeur retournée par ces actions pour décrire un nouvel état du gestionnaire est également de cette forme (voir exemples).

En sortie du dialogue, le résultat est donné par un couple *[Identificateur | ListeDesItemsSélectionnés]* ,
où *ListeDesItemsSélectionnés* est formée à partir de *ListeInit* et de *NoSelectionnés :*

:userItem(t, i, e, s)

Permet d'étendre les items traités par un dialogue. L'utilisateur doit alors fournir un paquet de règles ***<i,2>*** capable de répondre à un certain nombre d'actions requises par le gestionnaire de dialogue qui prend en charge la gestion des événements et des activations. Les paramètres ont la signification suivante :

t

Terme identifiant l'item en question et utilisé comme nom de zone si la valeur est extractible.

i

Identificateur d'une règle à deux arguments écrite par l'utilisateur pour gérer les actions concernant cet item (voir plus loin les actions possibles).

e

Valeur décrivant l'état de l'item et gérée par l'utilisateur lors des actions.

s

Décrit le type d'actions possibles sur cet item :

0 insensible.

1 activable par un clic.

2 activable par un clic ou une entrée de texte (TAB).

Les primitives d'agencement sont basées sur la notion de combinaison de rectangles: mettre deux rectangles en colonne définit un nouveau rectangle englobant les deux autres.

:col(l), ou **l**

Décrit une colonne alignée à gauche, dont le contenu est décrit par *l*. Si *l* est une liste, la primitive calcule la taille de chaque élément de la liste, et constitue une colonne d'éléments disposés les uns au dessus des autres, et alignés sur leur côté gauche.

© PrologIA

:ccol(l)

Décrit une colonne centrée verticalement, dont le contenu est décrit par la liste *l*.

:row(l)

Décrit une rangée horizontale décrite par la (liste de) primitive(s) *l*, et dont les éléments sont alignés sur leur bord supérieur: si *l* est une liste, la primitive calcule la taille de chaque élément de la liste, et constitue une rangée d'éléments disposés côte à côte, et alignés sur leur bord supérieur.

:crow(l)

Décrit une rangée d'objets décrits par *l* et centrés horizontalement, c'est à dire disposés de manière à ce que leurs points médians soient tous alignés sur une même horizontale.

:vfill(n)

Décrit un espacement vertical de *n* pixels, où *n* est un entier.

:hfill(n)

Décrit un espacement horizontal de *n* pixels, où *n* est un entier.

La primitive de groupement suivante permet de gérer l'activation ou la désactivation de zones entières pendant la saisie du dialogue.

:group([identificateur | valeur], DescriptionDesItems)

Permet d'associer un *identificateur* à un ensemble d'items regroupés géographiquement et sémantiquement.
valeur indique si le groupe est actif ou non (**0** pour inactif, **1** pour actif). Lorsque le groupe est inactif, il est dessiné en grisé et l'état de ses items ne peut être modifié. L'état d'un groupe ne peut être modifié dans la version actuelle que par les actions associées aux boutons. L'état d'un groupe n'est pas donné dans la liste de sortie, il est par contre fourni, dans la liste des états passée aux boutons, sous la forme d'une paire :

[identificateur | valeur]

Exemples:

```
> gr_dialog(<50,50>,
  :ccol([ "Voulez-vous recommencer le test?",
          :buttonD,
          :button("Annuler",fail)
        ] ),
  [],
  L );
```

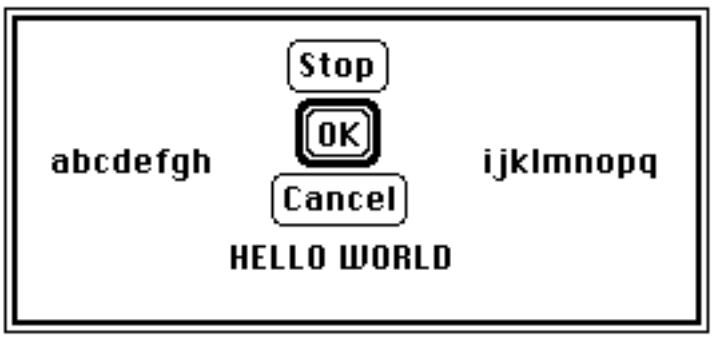
```
> gr_dialog(
  ["Choose the color you want",
  :crow([ :rb(<rb1,red>,1),"Red",
          :rb(<rb1,green>),"Green"] ),
  :crow([ :cb(cb1,0),"Extended colors"] ),
  :ccol([ :buttonD("Show Color"),
          :button("Cancel",fail)] )
  ],
  [[cb1|1]],
  [[cb1|C],[<rb1,red>|R],[<rb1,green>|G]] );
```


© PrologIA

Choose the color you want
Red
Green
Extended colors
Show Color
Cancel

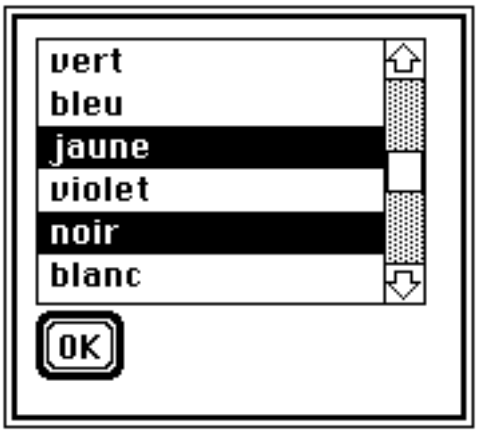
```
{ C = 1, R = 1, G = 0 }
>
```

```
>  gr_dialog(<50,50>,
     [:crow([
           "abcdefgh",
           :ccol([ :button("Stop",fail),
                   :buttonD("OK",[]),
                   :button("Cancel",fail),
                   "HELLO  WORLD"
                ] ),
           "ijklmnopq"
           ] )
     ],
   [],
   1 );
```




```
>  gr_dialog(
   [:glist(5,  10,  liste1,
       <[rouge,vert,bleu,jaune,
           violet,noir,blanc,orange],  [4,6],  2>
            ),
   :buttonD],
   [],
   L);
```


© PrologIA



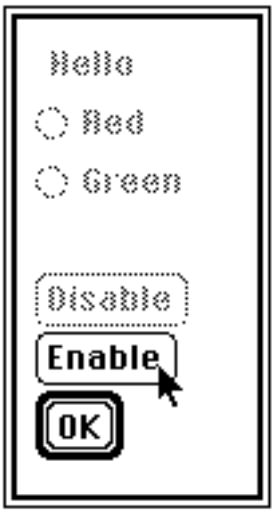

```
{ L = [[list1, jaune, noir]] }
>
```

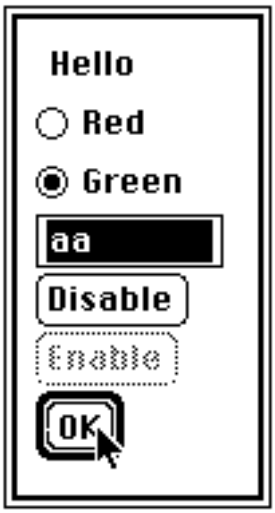
```
>
enable(l1, l3) ->
  set_item(l1, zone1, 1, l2)
  set_item(l2, zone2, 0, l3);

disable(l1, l3) ->
  set_item(l1, zone1, 0, l2)
  set_item(l2, zone2, 1, l3);

set_item([], i, b, []) ->
  block_exit(816, ["set_item, absent: "|i]);
set_item([[i,_]|l], i, b, [[i,b]|l]) -> ! ;
set_item([e|l], i, b, [e|l']) ->
  set_item(l, i, b, l') ;
```

```
> gr_dialog(
  [:group(  [zone1,0],
     [ "Hello",
       :crow([:rb1(<color,red>),"Red"]),
       :crow([:rb1(<color,green>,1),"Green"]),
       :editf(5,ed1,"aa"),
       :button2("Disable",disable)
     ] ),
    :group(  [zone2,1],
       :button2("Enable",enable)),
    :buttonD
  ],
  [],
  L);
```






```
{ L = [[color|green], [ed1|"aa"]] }
```


© PrologIA

Extensions utilisateur

Un *user_item* est un objet élémentaire apparaissant dans une boite de dialogue. L'utilisateur a la possibilité de définir ses propres items.

Il faut pour définir un *user_item*, lui donner un nom (*TermeIdentifiant*), une sensibilité (**0**, **1** ou **2**), la valeur initiale de son état, et lui associer un paquet des règles d'arité 2 . Ces 2 arguments seront respectivement la requête et la valeur de retour.
Ce paquet *<IdentificateurdeRègle, 2>* sert à préciser :

- la taille du rectangle englobant (requête *size*),
- le dessin correspondant à l'objet (requête *draw*)
- l'action associée à un clic (requête *click*)
- la réponse à la frappe de caractères (requête *edit*)
- l'information retournée au gestionnaire de dialogue en cas d'activation de cet item par un clic ou une édition (requête *extract*).

La sensibilité de l'item est définie de la façon suivante :

Sensibilité	Requête
0	size, draw.
1	size, draw, clic, extract.
2	size, draw, clic, edit, extract.

L'état d'un *user_item* est un terme structuré par l'utilisateur selon ses besoins.

:userItem(TermeIdentifiant, IdentificateurdeRègle, Etat, Sensibilité).

Au moment de l'appel, le premier argument de la règle ***<IdentificateurdeRègle, 2>*** contient la requête avec les paramètres pertinents extraits de l'item, et le deuxième argument est une variable libre qui devra être instanciée avec le résultat de la requête. Si la règle échoue, l'opération est ignorée.

Description de chacune des requêtes :

:size(TermeIdentifiant,Etat)

Demande de calcul de la taille de l'item lors de l'initialisation du dialogue. Le résultat doit être un doublet *<dx,dy>* en nombre de pixels.

:draw(TermeIdentifiant,Etat,Rectangle,Actif)

Demande de dessin de l'item *TermeIdentifiant* dans le rectangle *Rectangle* (de la forme *<x1,y1,x2,y2>*) avec la description d'état *Etat* et situé dans un groupe activé si *Actif* est **1**, désactivé si *Actif* est **0** (il est conseillé de ne pas montrer les valeurs de champ dans les zones désactivées). Le résultat est ignoré.

:click(TermeIdentifiant,Etat,Rectangle,PointCliqué)

Demande de réaliser l'action associée à un clic sur l'item. *PointCliqué* est de la forme *<x,y,m>* (voir ***<gr_click,4>)***.

Le résultat est un doublet *<NouvelEtat,Terminaison>* où :

NouvelEtat remplacera le champ *Etat* du *userItem*.

Terminaison sera de la forme :

<> Si aucune interaction n'a été faite avec l'utilisateur.

<n> Si la règle gère une boucle locale de traitement de l'item et que l'interaction se termine par un caractère RC (n=0) ou TAB (n=2) ou un clic non lu (n=1).

<x,y,m> Si la règle gère une boucle locale de traitement de l'item et que l'interaction se termine par un clic situé à l'extérieur du rectangle de l'item.

Si l'état de l'item change, il doit être redessiné avant la terminaison.

:edit(TermeIdentifiant,Etat,Rectangle)

Demande de saisie de caractères dans l'item. Le résultat sera de la même forme que ci-dessus.

© PrologIA

:extract(TermeIdentifiant,Etat)

Demande d'extraction de la valeur courante de l'item pour former le couple *[TermeIdentifiant | Valeur]* qui est fourni en sortie ou passé aux items avec des actions sur le dialogue (*:button2* par exemple). Si la requête échoue, on suppose qu'il n'y a pas de résultat à associer à l'item. Sinon le résultat doit être la partie valeur.

Exemple de création d'un item faisant défiler, lorsqu'on clique dessus, les différents éléments d'une liste :

```
>
circ_list(:size(_,<N,L>),  <x,20>)  ->
    max_width(L,  20,  x);
circ_list(:draw(_,V,R,  _actif),  _)  ->
    monItemDraw(V,  R,  _actif);
circ_list(:extract(_,<N,L>),  V)  ->
    arg2(N,  L,  V);
circ_list(:click(i,<N,L>,R,P),  <<N1,L>,<>>)  ->
    arg2(0,  L,  M)
    mod(N,  M,  N1-1)
    circ_list(:draw(i,<N1,L>,R,1),  _)  ;

max_width([],  X,  X+4)  ->;
max_width([S|L],  X,  X')  ->
    gr_stringwidth(S,  Y)
    max(X,  Y,  X0)
    max_width(L,  X0,  X');

maxof(X,Y,  X)  ->  !  {X>Y};
maxof(X,Y,  Y)  ->  ;
```

```
monItemDraw(<N,L>,  r,  _actif)  ->
     arg2(N,  L,  S)
     gr_moveto(x+2,  y1+14)
     gr_rect(:eraseRect,  r)
     gr_rect(:frameRect,  r)
     outItem(S,  _actif)
       {r  =  <x1,y1,x2,y2>};

outItem(S,  1)  ->  outml(S);
outItem(S,  0)  ->;

>  gr_dialog(
  [:row(
      [:userItem(  monItem(liste1),
               circ_list,
               <1,["FIRST","abcd","e","LAST"]>,
               ,1),
       "<--  user  Item"
      ] ),
  :buttonD
  ],
  [],
  L);
```

```
{ L = [[monItem(liste1) | "e"]] }
```


© PrologIA

Exemple de création d'un item affichant des icônes, et changeant d'icône lorsque l'on clique dessus :

```
>
iconManager(:size(_,_),  <32,32>)  ->  ;
iconManager(:draw(_,V,  R,  _),  _)  ->
     gr_icon(V,  R);
iconManager(:extract(_,V),  V)  ->  ;
iconManager(:click(N,V,R,P),  <V2,<>>)  ->
     mod(V+1,  3,  V2)
     iconManager(:draw(N,V2,R,1),  _  )  ;

>  gr_dialog(
    [:row( [ :userItem(myIcon,iconManager,1,1),
              "<--  user  Item"
            ] ),
    :buttonD  ],
   [],  L);
```

```
{ L = [[myIcon | 1]] }
```

9. Boutons actifs

Les primitives décrites dans cette section permettent d'associer des buts Prolog à des boutons, ces buts étant lancés dès que le bouton est pressé de manière similaire à l'activation d'un but par menu, c'est à dire en suspendant l'exécution en cours. Prolog III se charge du rafraîchissement du bouton.

X create_button(s,f,p,x1,y1)

Création d'un bouton de nom *s* dans la fenêtre de nom *f* (*s* et *f* doivent être des chaînes)

x1,y1

représentent les coordonnées du bouton par rapport au coin supérieur gauche de la fenêtre.

p

est l'identificateur du but (sans argument) qui sera lancé lorsque le bouton est cliqué.

X activate_button(s)

Rend activable le bouton de nom *s*.

X deactivate_button(s)

Rend inactif le bouton de nom *s*. Le texte est alors grisé.

X kill_button(s)

Destruction du bouton de nom s.

10. Primitives de manipulation de fichiers

input(x)

Avec x variable libre, affiche un dialogue de sélection d'un fichier (de type *texte* seulement) et unifie *x* avec le nom avec chemin du fichier choisi après l'avoir ouvert en lecture. Il y a échec si le bouton *Annuler* est cliqué. (voir aussi la primitive ***sfgetfile***).

© PrologIA

- **output(x)**

  Avec x variable libre, affiche un menu de dialogue de saisie de nom de fichier et unifie *x* avec le nom avec chemin du fichier saisi après l'avoir ouvert en écriture. Il y a échec si le bouton *Annuler* est cliqué. (voir aussi la primitive ***sfputfile***).

- **gr_load(s)**

  Dessine dans la fenêtre graphique courante le contenu d'un fichier graphique de nom *s* et de type MacPaint ou MacDraw :

  "PNTG" (MacPaint) ou "PICT" (MacDraw sauvé en format PICT). Il est possible de dessiner à des endroits différents des parties sélectionnées d'un fichier en jouant sur les primitives ***gr_setorigin*** et ***gr_rect(:clipRect,..)***.

- **sfgetfile(s)**

  Affiche une zone de dialogue permettant de sélectionner un fichier de type *texte* sur le disque, et unifie *s* avec le nom avec chemin de ce fichier (le fichier n'est pas ouvert). La règle échoue si le bouton *Annuler* est pressé. Cette règle est une forme simplifiée de la primitive suivante.

- **sfgetfile(point, L_types, filtre, s)**

  Affiche une zone de dialogue permettant de visualiser, parmi les fichiers se trouvant sur le(s) disque(s), ceux dont le type est dans *L_types*, et de sélectionner un fichier parmi ceux-ci (sans que le fichier ne soit ouvert).

  point

  est un point de format <*x*,*y*> indiquant l'endroit où est affiché le dialogue.

  L_types

  est une liste de types précisant les fichiers à proposer. Un type est une chaîne de 4 caractères tel que défini dans *Inside Macintosh*.

  Exemple : *["TEXT","APPL"]* montrera les fichiers de type *texte* et ceux de type *application*.

filtre

est une procédure de filtrage (en plus du type) pour l'acceptation des fichiers à proposer. Ce doit être *[]* pour l'instant.

s

est unifié avec la sélection faite par l'utilisateur. Il est de la forme : <*C*,*T*,*N*> où *C* est le *créateur* (chaîne de 4 caractères identifiant l'application qui a créé ce fichier), *T* est le type du fichier (chaîne de 4 caractères comme *"TEXT"*, *"APPL"*, ...), et *N* est une chaîne indiquant le chemin d'accès et le nom du fichier sélectionné (le fichier n'est pas ouvert).

La règle échoue si le bouton *Annuler* est pressé.

- **sfputfile(s)**

Affiche un dialogue permettant de saisir un nom de fichier sur le disque, et unifie *s* avec le nom avec chemin de ce fichier. Si l'utilisateur tape un nom de fichier déjà existant, une alerte le prévient. Le fichier donné par l'utilisateur n'est ni créé, ni ouvert. La règle échoue si le bouton *Annuler* est pressé.

- **sfputfile(point, prompt, nomInitial, s)**

Cette règle est une forme paramétrable de la précédente.

point

est un point de format <*x*,*y*> indiquant l'endroit où est affiché le dialogue.

prompt

est une chaîne, (éventuellement vide) contenant un message destiné à l'utilisateur.

nomInitial

est une chaîne (éventuellement vide) permettant de proposer un nom de fichier dans la zone d'édition. Ce nom ne doit pas comporter de chemin.

s

est unifié avec le nom avec chemin validé par l'utilisateur (le fichier n'est ni créé, ni ouvert).

La règle échoue si le bouton *Annuler* est pressé.

© PrologIA

Index des primitives graphiques

:button 501
:button2 501
:buttonD 502
:cb 500
:ccol 504
:clipRect 476
:col 504
:crow 504
:editf 501
:erase 475; 476; 477
:eraseOval 475
:eraseRect 475
:eraseRoundRect 475
:frame 475; 476
:frameOval 475
:frameRect 475
:frameRoundRect 475
:glist 502
:group 505
:hfill 505
:invert 475; 476; 477
:invertOval 475
:invertRect 475
:invertRoundRect 475
:paint 475; 476; 477
:paintOval 475
:paintRect 475
:paintRoundRect 475
:rb 500
:rb1 501
:row 504
:text 500
:userItem 503
:vfill 504
activate_button 515
add_stdmenu 497
"apple" 497
bitmap de rafraîchissement 468
boîte à cocher 500
bouton 485; 501
bouton de terminaison 502
check_item 496
clear_menubar 497
clear_window 472
clic 480; 486; 490; 491; 504; 510
cochage d'un item 496
command_menu 496
console 464
"control" 497
coordonnées horizontales 473
coordonnées verticales 473
crayon 474; 481; 482
create_button 515
create_window 471
deactivate_button 515
dialogue de saisie de nom de fichier 516
dialogue de sélection d'un fichier 516
"DISTANCE" 468
"EDIT" 467
"edit" 497
enable_menu 496
espacement horizontal 505
espacement vertical 504
fenêtre graphique courante 473
"file" 497
file_window 466
"find" 497

"FONT" 469; 471
fonte 472; 482; 483; 484
"FONTSIZE" 468; 471
front_window 466
get_key 487
get_screen 466
get_window 466
gr_arc 476
gr_arc' 477
gr_button_hit 486
gr_choosecolor 482
gr_click 479; 480
gr_color 481
gr_color2 481
gr_color3 481
gr_dialog 498
gr_draw_buttons 485
gr_editf 486
gr_erase 474
gr_font 484
gr_getmouse 480
gr_icon 478
gr_line 474
gr_lineto 474
gr_list 489
gr_load 516
gr_mode 482
gr_move 474
gr_moveto 474
gr_pen 482
gr_penloc 474
gr_polygon 476
gr_popupItem 491
gr_print 492
gr_rect 475
gr_setorigin 474
gr_stringwidth 482
gr_text 483
gr_window 472
gr_window_is 472
"GRAPHICS" 467
groupe 500
gtty 489
input 516
item coché 494
kill_button 515
kill_window 472
ligne de texte éditable 501
ligne de texte non éditable 500
load_graphic 464
MacDraw 516
MacPaint 516
menu fugitif 493
menu hiérarchique 493
menu pop-up 493
"MODAL" 467
modes de transfert 483
motif 482
"MPW" 489
new_window 467
"NLINES" 468
option fermeture 473
output 516
pixels 466; 471; 474; 482; 504; 511
police de caractères 469
primitive de groupement 505
primitives d'agencement 504
primitives de description 500
print_window 472
raccourci 496
radio-bouton 500
rafraîchissement de la fenêtre graphique 473
reset_window 472
restore_menubar 497
retour chariot 486; 490; 502; 511

© PrologIA

"SAVE" 468; 470
save_menubar 497
save_window 469
set_menu 493
set_window 470
sfgetfile 516; 517
sfputfile 517; 518
"SHAPE" 468
souris 480
srcBic 483
srcCopy 483
srcOr 483
srcXor 483
stty 488
style 482; 483; 497
style_menu 497
tabulation 486; 490; 504; 511
taille de la fonte 472; 482; 483; 484
taille de la police de caractères 468
"TTY" 467; 489
user_item 510
userItem 510
"window" 497

Association Prolog
HERITAGE

Annexes

A. Liste des messages d'erreur de Prolog III
B. Liste des règles prédéfinies par catégories
C. Quelques programmes Prolog III écrits en syntaxe "Edimbourg"

A. Liste des messages d'erreur de Prolog III

1: Debordement de l'espace de copie
2: Debordement de l'espace de backtracking
3: Debordement de l'espace auxiliaire
4: Debordement de l'espace auxiliaire
5: Debordement de l'espace auxiliaire
6: Debordement de l'espace auxiliaire
7: Debordement de l'espace auxiliaire
8: Debordement de l'espace auxiliaire
9: Debordement de l'espace des regles
10: Division par zero
11: Debordement de l'espace auxiliaire
12: Reel trop grand (1)
13: Reel trop grand (2)
14: Reel indefini (1)
15: Reel indefini (2)

© PrologIA

16: Fichier inexistant
17: Fin de fichier inattendue
18: Tete de regle incorrecte
19: Debordement de la pile de marquage
20: Debordement de l'espace auxiliaire (booleen)
21: Entier trop long
22: Predicat evaluable incorrect
23: Erreur de lecture
24: Nombre incorrect d'argument du predicat evaluable
25: Tuple trop grand (2)
26: Chaine trop grande
27: Identificateur trop long (1)
28: Probleme dans assert
29: Caractere incorrect
30: Conflit Pev Regle
31: Operande non arithmetique
32: Tuple trop grand (2)
33: Erreur de fermeture de fichier (1)
34: Erreur d'ouverture de fichier
35: Erreur de fermeture de fichier (2)
36 Plus de fichier d'entree
37: Trop de fichiers ouverts
38: Trop de fichiers de sortie
39: Plus de fichiers de sortie
40: Erreur dans assert ou in_term
41: Erreur dans in_term
41: Erreur dans assert (1)
42: Mode d'insertion inconnu
46: Regle deja definie
48: Debordement de l'espace auxiliaire (simplex)
49: Erreur lors du compactage de l'espace des regles
50: Block indefini
51: Memoire saturee (1)
52: Erreur d'ouverture du fichier Prolog3.log
53: Erreur dans assert (2)
54: Impossible d'ouvrir le fichier

© PrologIA

55: Impossible de fermer le fichier
56: Memoire saturee (2)
100: Debordement lors de la simplification (equations)
101: Debordement lors de la simplification (inequations)
102: Debordement lors de la simplification
103: Debordement lors de la simplification (booleen)
200: Memoire saturee (3)
201: Fin de liste attendue
202: C'est une liste infinie
203: Chaine attendue
204: Argument de mauvais type
205: Prefixe trop long
206: Memoire saturee (4)
207: Identificateur trop long (2)
208: Contexte deja ferme
209: Identificateur inconnu
210: Famille non fermee
211: Regle inexistante (1)
212: Regle inexistante (1)
213: Identificateur attendu (1)
214: Memoire sature (fermeture context)
215: Ce n'est pas une syntaxe d'identificateur
216: Ce n'est pas une syntaxe de prefixe
217: Ce n'est pas un identificateur
218: Dictionnaire plein
219: Reel strictement positif attendu
220: Reel positif ou nul attendu
221: Symbole inconnu
222: Erreur d'ouverture de fichier (save)
223: Erreur entree/sortie
224: Erreur entree/sortie
225: Erreur entree/sortie
226: Memoire saturee (5)
227: Memoire saturee (6)
228: Argument de mauvais type
229: Type non defini

230: Dictionnaire plein
231: Longueur inconnue
232: Tableau trop grand
233: Identificateur attendu (2)
234: Entier attendu
235: Tableau deja defini
236: Tableau non defini
237: Debordement de tableau
238: Pas d'identificateur avec ce prefixe
239: Fichier non chargeable
240: Erreur de chargement
241: Prefixe inexistant
242: Erreur de chargement (2)
243: Débordement de l'espace des regles
244: Fonction non evaluable
245: Erreur dans assign
246: Erreur dans val
247: Erreur dans string_ident
248: Variable PrologEdit non definie
249: Erreur dans end_module
250: Acces deja defini (PRO_BIND)
251: Acces non defini

400: Configuration inconnue
422: Memoire saturee (7)

500: Plus assez d'espace. PrologIII abandonne ...
501: Demarrage PrologIII impossible ...
502: Fichier d'erreur non trouvé !
503: Le systeme de contraintes de la requete est insatisfaisable.
504: Le systeme de contraintes de la regle est insatisfaisable.
505: Caractere illegal
506: Erreur dans insert ou assert ou in_term
507: Erreur interne (1)
508: Erreur interne (2)
509: Non implemente (1)

510: Erreur interne (3)

1001: Programme abandonne
1016: Interruption utilisateur
1026: Reel trop grand.

1101: Erreur d'ouverture du fichier d'entree
1105: Eof ou erreur sur l'entree standard
1111: Erreur d'ouverture du fichier de sortie
1115: Le fichier est deja ouvert pour une autre operation
1130: L'unite d'entree/sortie existe deja

1255: Doublet attendu pour gr_polygon
1262: Les arguments du premier tuple doivent etre compris entre 0 et 1

1304: Erreur systeme

1362: 'page' et 'set_cursor' desactives
1363: Warning: fichier 'termcap' non trouve
1364: Warning: terminal inconnu pour 'termcap'
1365: Warning: 'clear_screen' non valable pour ce terminal
1366: Warning: variable 'term' non definie
1367: Warning: 'cursor_motion' non defini
1372: Coordonnees graphiques incorrectes
1386: La fenetre associee au bouton n'existe pas
1387: Argument autorise dans une fenetre de dessin uniquement
1388: Unite inexistante ou de mauvais type
1389: Regle predefinie non implantee
1390: Ecran non initialise pour le graphisme
1391: Font non definie
1392: Type de fenetre non valide
1393: Option non valide a la creation de la fenetre
1394: Nombre maximal d'items atteint pour ce menu
1395: Depassement du nombre de niveaux autorises
1396: L'item n'existe pas

1436: Impossible d'ouvrir un nouveau descripteur d'unite
1438: Depasse le nombre d'unites d'e/s ouvertes possibles
1462: Nb de points maximal d'un polygone depasse
1471: 'string_term': chaine trop longue

1548: 'lkload' : descripteur non present dans le fichier
1700: Erreur de lecture dans un co_process
1701: Erreur d'ecriture dans un co_process

1815: Echec interne dans 'gr_dialog'
1816: Echec dans une regle appelee par 'gr_dialog'
1817: Item incorrect dans 'gr_dialog'
1818: 'gr_dialog' : erreur interne

© PrologIA

B. Liste des règles prédéfinies par catégorie

Règles prédéfinies Prolog III

Arbres, listes, chaînes et tuples

AAarg(N, T1, T2)
arg2(N, L, T)
arg3(N, T1, T2)
bound_conc(U1, U2, U3)
bound_size(T, N)
bound_tree(T)
bound_tree'(T)
conc3(U1, U2, U3)
conc_string(S1, S2, S3)
find_pattern(S1, S2, N)
known_part(U1, U2, U3)
size(U, N)
substring(S1, N1, N2, S2)

Contrôle

assign(I, T)
assign(tab(N), T)
block(E, B)
block_exit(E)
block(E, C, B)
block_exit(E, C)
bound(T)
cut
default(T1, T2)
dif(T1, T2)
eq(T1, T2)
fail
findall(v,p,l)
free(T)
freelabel(T)

freeze(T, P)
known(T)
not(T)
predefined(T)
repeat
val(T1, T2)

Contraintes numériques

bound_mult(N1, N2, N3)
div(N1, N2, N3)
enum(R)
enum(R, R1)
enum(R, R1, R2)
gcd(N1,N2,N3)
lcm(N1,N2,N3)
max_value(R1, R2)
min_value(R1, R2)
mod(N1, N2, N3)
mult(N1,N2,N3)
numden(R,N1,N2)
particular_value(R1, R2)
pgcd(A,B,P)
ppcm(A,B,P)
trunc(R, N)

Conversion

char_code(C, N)
list_string(L, S)
list_tuple(L, U)
split(U, L)
string_bool(S,B)
string_ident(P, S, I)
string_ident(S, I)
string_integer(S,N)
string_real(S,F)

Environnement

cpu_time(N)
edinburgh
edit(I)
exit
exit(S)
garbage_collection
get_config(S,V)
get_configuration(U)
getenv(S1,S2)
no_trace
prologIII
quit

© PrologIA

quit(N)
reset_cpu_time
resume_trace
save
save(L, F)
save(S)
set_config(S, V)
set_configuration(U)
state
suspend_trace
sys_command(S)
trace
trace_file(S)

Entrées/Sorties

close_input
close_input(F)
close_output
close_output(F)
echo
in_char(C)
in_char'(C)
in_ident(I)
in_integer(N)
in_real(R)
in_sentence(T1,T2,T3)
in_sentence'(T)
in_string(S)
in_term(T)
inl(S)
input(F)
input_is(F)
insert
insert(F)
line
next_char(C)
next_char'(C)
no_echo
out(T)
outc(T)
outl(T)
outm(S1, S2, …, SN)
outml(S1, S2, …, SN)
output(F)
output_is(F)
reinsert
reinsert(F)
reload(F, L)

Gestion des règles et des identificateurs

add_implicit(S1,S2)
assert(T, Q)
asserta(T, Q)
assertz(T, Q)
assert''(T,Q)
close_context_dictionary(S)
current_context(T0, T1, T2, T3)
current_context(T0)
current_predicate(<I, A>)
def_array(I, N)
dictionary
dictionary(L)
dictionary(M, L)
end_module
kill_module(S)
list
list(<I, A>)
list(<I, A>,N)
list([<I1,A1>,...,<In,An>])
list(T)
module
new
redef_array(tab,N)
remove_implicit(S1, S2)
retract(T,Q)
rule(N,AT, Q)
rule(N,T, Q)
rule(T, Q)
rule_nb(<I, A>, N)
set_context (I, S, S', D)
set_ident(I,N)
suppress(I)
suppress(I, A)
suppress(I, A, N)
undef_array(tab)

Vérification de types

bool(B)
char(C)
dot(T)
ident(I)
integer(N)
is_bool(B)
is_char(C)
is_ident(I)
is_leaf(T)
is_num(R)
is_tuple(U)
is_univ(T)
num(R)

rational(R)
real(F)
string(S)
tuple(U)

Règles prédéfinies Edimbourg

Arbres, listes, chaînes et tuples

'='(X, Y)
'=..'(X, Y)
'=='(X, Y)
'\=='(X, Y)
functor(T, E, N)
member(X, L)
name(X, L)
op(N, I1, S)
op(N, I1, S, I2)

Contrôle

','(X, Y)
'->'(Y)
';'(X; Y)
'[]'(X, Y)
[]
[X | Y]
call(P)
nonvar(T)
true
var(T)

Evaluation de fonctions prédéfinies

'<'(X, Y)
'=:='(X, Y)
'=<'(X, Y)
'=\='(X, Y)
'>'(X, Y)
'>='(X, Y)
is(X, T)

Entrées/Sorties

consult(S)
get(X)
get0(X)
nl
put(X)

read(T)
see(F)
seen
tab(N)
tell(F)
told
write(T)
writeq(T)

Gestion des règles et des identificateurs

abolish(<I, A>)
abolish(I)
assert(T)
asserta(T)
assertz(T)
clause(T, Q)
listing
listing(<I, A>)
listing(I)
retract(X)
retractall(T)

Vérification de types

atom(T)
atomic(T)
number(R)

© PrologIA

C. Quelques programmes Prolog III écrits en syntaxe“Edimbourg”

1. Un petit exemple : menu.p3e
2. Calculs sur les grands nombres : fact.p3e
3. Calculs bancaires : bank.p3e
4. Manipulation d'arbres : leaves.p3e
5. Un peu de logique : god.p3e
6. Un problème de Lewis Caroll : lewis.p3e
7. Détection de pannes dans un additionneur : circuit.p3e
8. Un puzzle numérique : send.p3e
9. Un problème de remplissage : rectangle.p3e

Les programmes d'exemples de cette annexe sont des traductions en syntaxe Edimbourg des principaux programmes donnés dans ce manuel.

1. Un petit exemple : menu.p3e

Le programme :

```
%%% Le fameux repas equilibré .

repasLeger(H,P,D) :-
   horsDoeuvre(H,I),
   plat(P,J),
   dessert(D,K)
   {I>=0, J>=0, K>=0, I+J+K=<10}.

plat(P,I) :- viande(P,I).
plat(P,I) :- poisson(P,I).

horsDoeuvre(radis,1).
horsDoeuvre(pate,6).

viande(boeuf,5).
viande(porc,7).

poisson(sole,2).
poisson(thon,4).

dessert(fruit,2).
dessert(glace,6).
```

Une Question :

```
?- repasLeger(H,P,D).
```


© PrologIA

2. Calculs sur les grands nombres : fact.p3e

Le programme factorielle :

```
%%% Factorielle    N! = N*(N-1)*(N-2)*....*2*1

factorielle(0, 1).
factorielle(N, N*M) :- factorielle(N-1, M) {N >= 1}.
```

Questions (la 4ème question peut ne pas marcher avec les nombres flottants, selon l'implémentation des réels utilisée sur votre machine)

```
?- factorielle(10, X).

?- factorielle(40, X).

?- factorielle(100, X).

?- factorielle(100.0, X).

?- factorielle(1000, X).
```

3. Calculs bancaires : bank.p3e

Le programme :

```
/* Calcul bancaire. Le taux d'interêt est de 10% */

versementsCapital(<>, 0).
versementsCapital(<I>.X, C) :-
     versementsCapital(X, C+(10/100)C-I).
```

Quelques questions posées avec des nombres entiers ou flottants :

```
?- versementsCapital(<I, 2I, 3I>, 1000).

?- versementsCapital(<I, I, I>, 10000).

?- versementsCapital(<I, 2I, 3I>, 1000.0).

?- versementsCapital(<I, I, I>, 10000.0).
```


© PrologIA

4. Manipulation d'arbres : leaves.p3e

Le programme :

```
%%% Construit la liste des feuilles d'un arbre.

leavesOf(A, X) :-
     leavesOfTree(A, X, <>).

leavesOfTree(E[<>], <E>.X, X) .
leavesOfTree(E[U], X, X') :-
     leavesOfList(U, X, X')
     {U # <>}.

leavesOfList(<>, X, X) .
leavesOfList(<A>.U, X, X'') :-
     leavesOfTree(A, X, X'),
     leavesOfList(U, X', X'').
```

Questions :

```
?- leavesOf(measures("Max", <1+75/100, metres>, 1'), X).

?- leavesOf(weighs("Max",<2 P + 1, kg>,male(1'),`o`,'H'),X).
```

5. Un peu de logique : god.p3e

Le programme :

```
valueOfSomethingHasAlwaysExisted(B)
   { A = 1',
     A => (B | C) & ~(B & C),
     A => (D | E) & ~(D & E),
     D => B,
     E => ~C }.
```

LA question :

```
?- valueOfSomethingHasAlwaysExisted(B).
```


© PrologIA

6. Un problème de Lewis Caroll : lewis.p3e

Le programme :

```
%%%% Peut-on être député ? .

possibilite(X) :- casPossible(Y), sousEnsemble(X,Y).

sousEnsemble(<>,Y).
sousEnsemble(<E>.X,Y) :- elementDe(E,Y), sousEnsemble(X,Y).

elementDe(E,<E>.Y).
elementDe(E,<F>.Y) :- elementDe(E,Y) {E#F}.

casPossible(<
 <A,"avoir l'esprit clair">,
 <B,"avoir recu une bonne education">,
 <C,"discourir sans cesse">,
 <D,"employer son influence a des fins meritoires">,
 <E,"etre affiche dans les vitrines">,
 <F,"etre apte a etre depute">,
 <G,"etre un bienfaiteur du peuple">,
 <H,"etre digne d'eloges">,
 <I,"etre populaire">,
 <J,"etre sans pretention">,
 <K,"etre une femme">,
 <L,"laisser un souvenir imperissable">,
 <M,"posseder une influence">,
 <N,"savoir garder un secret">,
 <O,"s'exprimer bien">,
 <P,"valoir son pesant d'or">>)
{
 (F & ~C) => G,
 (A & O) => B,
 (K & H) => N,
 (G & ~D)=> ~F,
 (P & H) => J,
 (G & D) => H,
 (~I & ~P) => ~N,
 (C & F) => H,
 (N & J) => (G & L),
 (K & G) => I,
 (P & C & L) => E,
 (K & ~A & ~B) => ~F,
```

```
  (N & ~C) => ~I,
  (A & M & D) => G,
  (G & J) => ~E,
  (N & D) => P,
  (~O & ~M) => ~K,
  (I & H) => (G | J)
}.
```

Exemples de questions permettant de découvrir des liens (s'il en existe) entre des propriétés :

```
?- possibilite(<<A,"avoir l'esprit clair">,
               <I,"etre populaire">,
               <N,"savoir garder un secret">>).

?- possibilite(<<N,"savoir garder un secret">,
               <F,"etre apte a etre depute">,
               <P,"valoir son pesant d'or">>).

?- possibilite(<<K,"etre une femme">,
               <F,"etre apte a etre depute">>)
     {K!bool,F!bool}.
```


© PrologIA

7. Détection de panne dans un additionneur : circuit.p3e

Le programme :

```
/*Détecteur de pannes
Notez que la 3ème contrainte de la règle suivante
contient le terme (U1|U2), parenthésé afin de pallier
une erreur dans l'analyse de cet opérateur OU (|) .
*/

circuit(<X1,X2,X3>, <Y1,Y2>, <P1,P2,P3,P4,P5>) :-
     atMostOneTrue(<P1,P2,P3,P4,P5>)
     { ~P1 => ( U1 <=> X1 & X3 ),
            ~P2 => ( U2 <=> X2 & U3 ),
            ~P3 => ( Y1 <=> (U1 | U2) ),
            ~P4 => ( U3 <=> ~(X1 <=> X3)),
            ~P5 => ( Y2 <=> ~(X2 <=> U3))  }.

atMostOneTrue(P) :- orOnAtMostOneTrue(P,_p).

orOnAtMostOneTrue(<>, 0').
orOnAtMostOneTrue(<E>.P,  E|V) :-
     orOnAtMostOneTrue(P, V)
     {E&V = 0'}.

%%% Ces règles sont utilisées par le second jeu de questions:
booleans(<>).
booleans(<B>.L) :- boolean(B), booleans(L).

boolean(0').
boolean(1').
```

Premier jeu de questions :

```
?- circuit(<1',1',0'>,<0',1'>,<P1,P2,P3,P4,P5>).

?- circuit(<0',0',1'>,<0',1'>,<P1,P2,P3,P4,P5>).

?- circuit(<1',0',1'>,<0',0'>,<P1,P2,P3,P4,P5>).

?- circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,1',P4,P5>).
```

Second jeu de questions :

```
?- circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,1',P4,P5>),
       booleans(<X1,X2,X3,Y1,Y2>).

?- circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,1',P4,P5>),
      booleans(<X1,X2,X3,Y1,Y2>),
       circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,0',P4,P5>).

?- circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,1',P4,P5>),
      booleans(<X1,X2,X3,Y1,Y2>),
       not(circuit(<X1,X2,X3>,<Y1,Y2>,<P1,P2,0',P4,P5>)).
```


© PrologIA

8. Un puzzle numérique : send.p3e

Le programme :

```
/* L'addition doit être résolue avec des chiffres distincts :

    SEND
  + MORE
  ------
   MONEY
*/

solution(I, J, K) :-
   allDifferentDigits(<M,S,O,E,N,R,D,Y>),
   integers(<M,S,O,E,N,R,D,Y>)
   { S#0, M#0, I+J=K,
     I =          1000S + 100E + 10N + D ,
     J =          1000M + 100O + 10R + E ,
     K = 10000M + 1000O + 100N + 10E + Y
     }.

integers(<>).
integers(<E>.R) :- enum(E), integers(R).

allDifferentDigits(<>).
allDifferentDigits(<X>.S) :-
     outsideOf(X, S),
     allDifferentDigits(S)
     {0 =< X, X =< 9}.

outsideOf(X, <>).
outsideOf(X, <Y>.S) :- outsideOf(X, S) {X#Y}.
```

La question :

```
?- solution(I,J,K).
```

9. Un problème de remplissage : rectangle.p3e

Le programme :

```
%%% Remplir un rectangle 1 x A avec N carrés distincts .

remplirRectangle(A, C) :-
   creerCarres(C),
   remplirZone(<-1,A,1>, L, C, <>)
   {A >= 1}.

creerCarres(<>).
creerCarres(<B>.C) :-
   creerCarres(C),
   rendreDistinct(B, C)
   {B > 0}.

rendreDistinct(B, <>).
rendreDistinct(B, <B'>.C) :-
   rendreDistinct(B, C)
   {B # B'}.

remplirZone(<V>.L, <V>.L, C, C)
   {V >= 0}.
remplirZone(<V>.L, L''', <B>.C, C'' ) :-
   placerCarre(B, L, L'),
   remplirZone(L', L'', C, C'),
   remplirZone(<V + B, B>.L'', L''', C', C'')
   {V < 0}.

placerCarre(B, <H, 0, H'>.L, L') :-
   placerCarre(B, <H + H'>.L, L')
   {B > H}.
placerCarre(B, <H, V>.L, <-B + V>.L)
   {B = H}.
placerCarre(B, <H>.L, <-B , H - B>.L)
   {B < H}.
```

Une question : comment remplir un rectangle de taille inconnue avec 9 carrés différents et de tailles inconnues ?):

```
?- remplirRectangle(A, C)  {C :: 9}.
```


© PrologIA

Index

! (coupure) 164
, 418
-> 417
| (coupure) 164
< 411
<> (tuple vide) 27
= 412
=.. 410
=:= 415
=< 411
== 413
=\= 416
> 411
>= 411
[ ] (liste vide) 33
[ ] (symbole fonctionnel) 83
[] 420
\== 414

A

abandon de l'exécution d'un but (block_exit) 166
abolish 377
abs 172; 369
activate_button 515
add 170; 368
add_implicit 183; 202
add_stdmenu 497
addition 30; 98
affectation 47
affectation (assign) 173
affectation booléenne 125
ajout d'une règle (assert) 186
"apple" 497
arbres 20; 70
arbres factuels 71
arbres infinis 75
arg 81; 203
arg2 84; 204
arg3 80; 205
arithmétique 14
assert 186; 206; 378
assert'' 187; 208
asserta 186; 207; 380
assertz 187; 208; 378
assign 173; 174; 209
atan 172; 370
atom 381
atomic 382

B

bitmap de rafraîchissement 468
block 166; 212
block_exit 166; 212
boîte à cocher 500
bool 129; 214
booléens 16; 118
  syntaxe 428
bouclage 162
bound 149; 167; 215
bound_conc 80; 216
bound_mult 105; 156; 217
bound_size 80; 151; 218
bound_tree 219
bound_tree' 220
bouton 485; 501

bouton de terminaison 502
buts 60
syntaxe 440

C

call 383
caractères 25
caractère d'échappement (\) 430
caractère désigné 430
espace 428
remarques 441
syntaxe 431
chaînes de caractères 13; 27; 71; 85
syntaxe 431
changement de signe 30
char 221
char_code 222
chargement d'un programme (input) 8
check_item 496
clause 384
clear_menubar 497
clear_window 472
clic 480; 486; 490; 491; 504; 510
close_context_dictionary 183; 223
close_input 9; 195; 224
close_output 11; 197; 225
cochage d'un item 496
command_menu 496
commentaire 442; 459
comparaisons numériques 52
conc3 80; 154; 226
conc_string 85; 227
concaténation 12; 76
retardée 154
connu (terme) 148
console 6; 464
constantes 23
booléennes 119
numériques 97
syntaxe 428
constructeur
d'arbres 72
de tuples 72
général d'arbres 72
consult 385
contexte courant 179
contexte de lecture-écriture 24; 179
contraintes 54; 438
booléennes 16; 120
exemples 122
forme normale 123
simplification 127
contrainte de taille 76
numériques 101
exemples 101
forme normale 102
retardées 156
retardées 151
syntaxe 439
contraintes de type
!bool 53; 120
!boolt 53; 120
!char 53
!chart 53
!id 53
!idt 53
!num 53; 100
!numt 53; 100
!tuple 53
contraintes numériques 14
"control" 497
contrôle 162
coordonnées horizontales 473
coordonnées verticales 473
cos 172; 370
coupure 164
cpu_time 198; 228
crayon 474; 481; 482
create_button 515
create_window 471
current_context 182; 229
current_predicate 187; 230
cut 231
cut (coupure) 164

D

def_array 174; 232
default 167; 234

© PrologIA

définition de contexte (set_context) 180
deactivate_button 515
dialogue de saisie de nom de fichier 516
dialogue de sélection d'un fichier 516
dictionary 184; 235
dif 237
"DISTANCE" 468
div 106; 171; 238; 368
division 31; 98
dot 239

E

echo 197; 240; 341
edinburgh 241
"EDIT" 467
"edit" 497
edit 242
égalité 50
enable_menu 496
end_module 185; 243
ensemble de contraintes
  booléennes 122
entrée-sorties 192
enum 104; 244
epsilon_float 342
eq 246
eql 171; 369
escape 431
espacement horizontal 505
espacement vertical 504
et 29; 119
exécution d'un programme 64
exemples
  arbres 71
  caractères 40
  chaînes 41
  constructeurs 73
  contraintes 56
  contraintes booléennes 122
  contraintes numériques 101
  contraintes sur les arbres 74
  contraintes sur les chaînes 85
  contraintes sur les tuples 78
  entiers 39
  expressions booléennes 43; 120
  expressions numériques 42; 99
  flottants 40
  freeze 149
  identificateurs 38
  listes 84
  modules 186
  règles 61
  requêtes 61
  termes 38; 45
  utilisation de la coupure (/) 164
exemples de programmes
  additionneur binaire 135
  au plus un vrai 131
  calcul bancaire 108
  calcul des feuilles 87
  ChatsOiseaux 15
  crible d'Eratosthène 90
  element_de 13
  K éléments vrais 133
  naive reverse 82
  puzzle logique 143
  quick sort 87
  remplissage d'un rectangle 110
  RepasLeger 9
  SEND + MORE = MONEY 109
  suite périodique 89
  un ou sur 2 variables 129
  un ou sur une liste 130
exit 198; 247
exp 172; 370
expressions
  booléennes 43; 119; 437
  numériques 42; 97; 436
    maximum 105
    minimum 105
    prédicat val 168

F

fail 249
faits vrais 62
famille 178
  partie fermée 178
Fdecimal 108

fenêtre graphique courante 473
feuille 21
Fexact 108
Ffloat 108
"file" 497
file_window 466
fin de session (quit) 11
"find" 497
find_pattern 85; 251
findall 250
Fint+fract 108
Fnormal 108
fonctions évaluables 170
"FONT" 469; 471
fonte 472; 482; 483; 484
"FONTSIZE" 468; 471
format_decimal 342
format_float 342
format_out_num 108; 341
  Fdecimal 341
  Fexact 341
  Ffloat 341
  Fint+fract 341
  Fnormal 341
formats d'entrée-sorties 108
forme clausale 124
forme normale
  contraintes booléennes 123
  contraintes numériques 102
forme normale conjonctive 123
formfeed 431
free 149; 168; 252
free_label 253
freeze 157; 254
front_window 466
functor 386

G

garbage_collection 255
gcd 256
get 387
get0 388
get_config 257
get_configuration 258
get_key 487
get_screen 466
get_window 466
getenv 259
gr_arc 476
gr_arc' 477
gr_button_hit 486
gr_choosecolor 482
gr_click 479; 480
gr_color 481
gr_color2 481
gr_color3 481
gr_dialog 498
gr_draw_buttons 485
gr_editf 486
gr_erase 474
gr_font 484
gr_getmouse 480
gr_icon 478
gr_line 474
gr_lineto 474
gr_list 489
gr_load 516
gr_mode 482
gr_move 474
gr_moveto 474
gr_pen 482
gr_penloc 474
gr_polygon 476
gr_popupItem 491
gr_print 492
gr_rect 475
gr_setorigin 474
gr_stringwidth 482
gr_text 483
gr_window 472
gr_window_is 472
"GRAPHICS" 467
groupe 500
gtty 489

I

ident 260
identificateurs 24; 433; 444

© PrologIA

abrégés 177
syntaxe 177; 434; 444
complets 177
séparateur 177
syntaxe 177; 434; 444
if 172; 369
implique (opération) 29; 119
implique (relation) 52; 120
in_char 193; 261
in_char' 262
in_ident 263
in_integer 264
in_real 265
in_sentence 194; 266
in_sentence' 267
in_string 268
in_term 269
inégalité 50; 101
inéquations numériques 104
inf 171; 369
infe 171; 369
inférieur ou égal 101
inférieur strict 101
inl 270
input 8; 194; 271; 516
input_is 272
insert 188; 189; 273
integer 106; 275
interruption 167
is 389
is_bool 129; 276
is_char 277
is_ident 278
is_leaf 279
is_num 106; 280
is_tuple 81; 281
is_univ 282
item coché 494

J

jeu de caractères 427

K

kill_button 515
kill_module 12; 191; 283
kill_window 472
known 148; 284
known_part 81; 152; 285

L

lcm 286
ligne de texte éditable 501
ligne de texte non éditable 500
line 197; 287
linefeed 431
list 7; 9; 11; 189; 288
list_string 86; 293
list_tuple 84; 294
liste vide 84
syntaxe 429
listes 33; 83
listing 390
littéral 124
ln 172; 370
load_graphic 464

M

MacDraw 516
machine Prolog III 65; 162
MacPaint 516
max_value 105; 295
maximum d'une expression numérique 105
member 391
menu fugitif 493
menu hiérarchique 493
menu pop-up 493
min_value 105; 296
minimum d'une expression numérique 105
mod 106; 171; 297; 369
"MODAL" 467
modes de transfert 483
module 24; 178; 298
objet 186

source 185
module(directive) 185
modules
exemples 186
moins unaire 98
motif 482
"MPW" 489
mul 170; 368
mult 105; 157; 158; 299
multiplication 31; 98

N

naive_reverse 82
name 392
new 300
new_window 467
newline 431
next_char 301
next_char' 302
nil 33
nl 393
"NLINES" 468
no_echo 197; 303
no_trace 304
nombres 14
entiers 26
syntaxe 428
flottants 26; 94
syntaxe 429
fractionnaires 94
rationnels 94
non 29; 119
non (prédicat) 166
nonvar 394
not 305
num 106; 306
number 395
numden 307

O

omission de l'opérateur de multiplication 99
op 396
opérateurs 445
booléens
équivalent 119
et 119
implique 119
non 119
ou 119
Edimbourg 459
numériques 98
opérations 28
arithmétiques 30; 98
booléennes 29
concaténation de tuples 32; 76
construction d'arbre 31; 71
construction de tuple 32; 71
construction générale d'arbre 32; 71
option fermeture 473
ou 29; 119
out 195; 308
outc 196; 309
outl 195; 310
outm 196; 311
outml 7; 196; 312
output 11; 197; 313; 516
output_is 314

P

paire pointée 33; 83
paramètres graphiques
:button 501
:button2 501
:buttonD 502
:cb 500
:ccol 504
:clipRect 476
:col 504
:crow 504
:editf 501
:erase 475; 476; 477
:eraseOval 475
:eraseRect 475
:eraseRoundRect 475
:frame 475; 476
:frameOval 475

© PrologIA

:frameRect 475
:frameRoundRect 475
:glist 502
:group 505
:hfill 505
:invert 475; 476; 477
:invertOval 475
:invertRect 475
:invertRoundRect 475
:paint 475; 476; 477
:paintOval 475
:paintRect 475
:paintRoundRect 475
:rb 500
:rb1 501
:row 504
:text 500
:userItem 503
:vfill 504
parasite 164
particular_value 105; 315
partie fermée d'une famille 179
pgcd 316
pixels 466; 471; 474; 482; 504; 511
plus unaire 98
police de caractères 469
ppcm 317
précision parfaite 26; 94
predefined 190; 318
préfixe 24
syntaxe 177; 434; 444
primitive de groupement 505
primitives
booléens 128
conversions de chaînes 86
numérique 104
tuples 80
primitives d'agencement 504
primitives de description 500
print_window 472
priorités
des opérateurs booléens 119
des opérateurs numériques 98
projection sur un sous-vocabulaire 127
prologIII 319
put 397

Q

qualifieur 24
quit 8; 198; 320

R

raccourci 496
rad 172; 370
radio-bouton 500
rafraîchissement de la fenêtre graphique 473
rational 106; 322
read 398
real 106; 323
récapitulatif
opérations et relations booléennes 123
opérations et relations définies sur les arbres et les tuples 79
redef_array 324
règles 60
prédéfinies
booléens 128
numérique 104
tuples 80
syntaxe 440; 456
reinsert 188; 189; 325
relations 49
booléennes 120
comparaisons numériques 52
de type 53
égalité 50
implication 52; 120
inégalité 50
nombre de fils 74
numériques 100
unaires 53
reload 191; 327
remove_implicit 183; 328
repeat 329
requêtes 61
syntaxe 440; 456
reset_cpu_time 198; 330
reset_window 472

résolution des systèmes de contraintes 57
restore_menubar 497
restrictions
concaténation 77
linéarité 102
nombres entiers 102
taille 76
resume_trace 331
retardements
concaténation 154
contraintes non-linéaires 106
contraintes numériques 156
freeze 149
tailles 151
retour chariot 486; 490; 502; 511
retract 190; 332; 399
retractall 400
return 431
rubout 431
rule 190; 333
rule_nb 336

S

sauvegarde d'un module 11
sauvegarde de modules (save) 192
"SAVE" 468; 470
save 192; 337
save_menubar 497
save_window 469
see 401
seen 402
sémantique 62
déclarative 62
des termes 46
opérationnelle 62
séparateur 177
set_config 108; 340
echo 341
epsilon_float 342
format_decimal 342
format_float 342
format_out_num 341
statistics 342
syntax 342
trace 340
tty_wrap 342
undefined_rule 343
set_configuration 344
set_context 180; 182; 345
set_ident 346
set_menu 493
set_window 470
sfgetfile 516; 517
sfputfile 517; 518
"SHAPE" 468
si_alors_sinon 166
simplification
contraintes booléennes 127
sin 172; 370
size 80; 152; 347
solution d'un système de contraintes 57
booléennes 126
souris 480
soustraction 31; 98
split 81; 348
sqrt 173; 370
srcBic 483
srcCopy 483
srcOr 483
srcXor 483
state 349
string 87; 350
string_bool 86; 351
string_ident 86; 352
string_integer 86; 354
string_real 86; 355
stty 488
style 482; 483; 497
style_menu 497
sub 170; 368
substring 86; 356
sup 171; 369
supe 172; 369
supérieur ou égal 101
supérieur strict 101
suppress 10; 15; 191; 357
suppression
de règles (suppress) 10; 191
suspend_trace 360
symboles relationnels

© PrologIA

booléens 120
numériques 100
syntax 342
Edinburgh 342
Prolog III 342
syntaxes
choix du mode syntaxique 424
syntaxe de base 424
syntaxe Edimbourg 425
sys_command 361
systèmes de contraintes 54
syntaxe 438; 454

T

tab 403
tableaux (def_array) 174
tabulation 431; 486; 490; 504; 511
taille de la fonte 472; 482; 483; 484
taille de la police de caractères 468
tailles 76
retardées 151
tan 173; 370
tell 404
termes 35
connus 148
sans variable 37
syntaxe 435; 447; 450
tête de règle 60
told 405
trace 340; 362
trace_file 363
true 406
trunc 106; 172; 364; 369
"TTY" 467; 489
tty_wrap 342
tuple 81; 365
tuples 12; 22; 71; 76

U

undef_array 366
undefined_rule 343
error 343
fail 343
warning 343
unités d'entrée/sortie 192
user_item 510
userItem 510

V

val 169; 367
abs 369
add 368
atan 370
cos 370
div 368
eql 369
exp 370
if 369
inf 369
infe 369
ln 370
mod 369
mul 368
rad 370
sin 370
sqrt 370
sub 368
sup 369
supe 369
tan 370
trunc 369
valeurs booléennes 25
var 407
variables 34
d'écarts 104
syntaxe 433; 443

W

"window" 497
write 408
writeq 409

Association Prolog
HERITAGE

Prolog III

Version 1.3

Manuel d'Utilisation

Macintosh

Décembre 1990

© PrologIA

Association Prolog
HERITAGE

Garantie et responsabilités

PrologIA n'offre aucune garantie, expresse ou tacite, concernant ce manuel ou le logiciel qui y est décrit, ses qualités, ses performances ou sa capacité à satisfaire à quelque application que ce soit.

PrologIA ne pourra être tenue responsable des préjudices directs ou indirects, de quelque nature que ce soit, résultant d'une imperfection dans le programme ou le manuel, même si elle a été avisée de la possibilité que de tels préjudices se produisent. En particulier, elle ne pourra encourir aucune responsabilité du fait des données mémorisées ou exploitées, y compris pour les coûts de récupération ou de reproduction de ces données.

L'acheteur a toutefois droit à la garantie légale dans les cas et dans la mesure seulement où la garantie légale est applicable nonobstant toute exclusion ou limitation.

Droits d'auteur

Ce manuel et le logiciel qu'il décrit sont protégés par les droits d'auteur. Au terme de la législation traitant de ces droits, ce manuel et ce logiciel ne peuvent être copiés ou adaptés, en tout ou en partie, sans le consentement écrit de PrologIA, sauf dans le cadre d'une utilisation normale ou pour faire une copie de sauvegarde. Ces exceptions n'autorisent cependant pas la confection de copies à l'intention d'un tiers, que ce soit ou non pour les vendre.

Prolog III est une marque déposée de PrologIA.

Association Prolog
HERITAGE

Utilisation de Prolog III sur Macintosh

1. **Installation de Prolog III**
2. **Utilisation de Prolog III**
3. **Spécificités de Prolog III Macintosh**
4. **A propos de l'environnement**
5. **Ajout de règles**
6. **Appel de Prolog III depuis le langage C**

Que trouve-t-on dans ce chapitre ?

Ce chapitre explique toutes les caractéristiques du système Prolog III qui dépendent de la machine utilisée ou de son système d'exploitation.
Il y est question principalement de la procédure d'installation du système Prolog III et des particularités de son activation (paramètres d'exécution, tailles des espaces, etc…).
Ce chapitre explique aussi comment ajouter à Prolog III, à travers C, vos propres règles prédéfinies externes, écrites dans n'importe quel langage de programmation.

Décembre 1990

© PrologIA

1. Installation de Prolog III

Ce paragraphe décrit la procédure d'installation du logiciel Prolog III sur Apple Macintosh. Il est recommandé de le lire en entier avant de commencer l'installation.

Matériel et logiciel requis

Prolog III est disponible sur toute la gamme Macintosh, du Mac Plus au Mac II. Il faut au minimum 2,5 Mo[1] de mémoire vive; 4 Mo ou plus sont souhaitables pour une utilisation confortable. Prolog III est compatible avec MultiFinder et peut fonctionner en tâche de fond. Bien qu'il ne soit pas impossible de travailler avec deux disquettes, un disque dur est fortement conseillé.

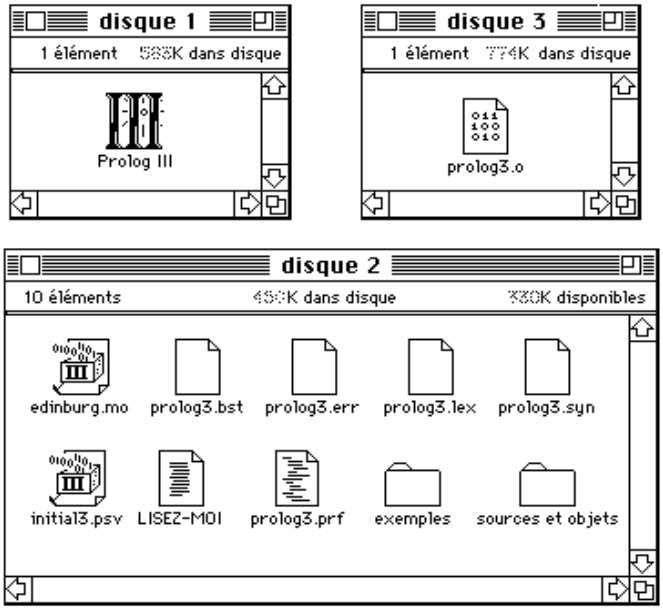
Pour ajouter à Prolog III des règles prédéfinies implantées par des sous-programmes externes, vous aurez besoin en outre de l'environnement de programmation MPW Version 2 d'Apple, avec le compilateur C et les logiciels et bibliothèques qui lui sont associés, ou bien le compilateur de tout autre langage compatible avec MPW C (c'est-à-dire, permettant les appels inter-langages de sous-programmes). Ici, la taille des fichiers objets et de l'application à construire impose la présence d'un disque dur.

Contenu du kit d'installation

Le kit de distribution de Prolog III se compose de trois disquettes de 800 Ko. Les « disque 1 » et « disque 2 » contiennent surtout les fichiers dont vous avez besoin pour *utiliser* Prolog III. Le dossier « sources et objets » du « disque 2 » et « disque 3 » contiennent les fichiers requis pour *augmenter* Prolog III par l'ajout de vos propres règles prédéfinies écrites en C ou tout autre langage, ou bien *augmenter* votre application par des appels Prolog :

1 Avec un tout petit système, et sans multifinder.

© PrologIA



Fichiers indispensables pour utiliser Prolog III

`Prolog III`

Fichier exécutable contenant l'interprète de Prolog III. C'est une application Macintosh « ordinaire » que l'on démarre à la souris par un double clic.

`prolog3.bst`

Fichier nécessaire au démarrage de l'interprète.

`prolog3.lex`

Fichier de données pour l'analyseur syntaxique de Prolog III.

`prolog3.sya`

Messages d'erreur de l'analyseur (version anglaise).

`prolog3.syf`

Messages d'erreur de l'analyseur (version française).

`prolog3.syn`

Messages d'erreur de l'analyseur. Il s'agit en fait d'une copie de l'un des fichiers *prolog3.sya* ou *prolog3.syf*.

`prolog3.era`

Messages d'erreur de l'interprète (version anglaise).

`prolog3.erf`

Messages d'erreur de l'interprète (version française).

prolog3.err

Messages d'erreur de l'interprète. Il s'agit en fait d'une copie de l'un des fichiers *prolog3.era* ou *prolog3.erf*.

initial3.psv

Etat sauvé initial : contient notamment les déclarations, sous une forme codée, des règles prédéfinies.

prolog3.prf

Fichier de préférences : contient une configuration sous la forme d'une ligne de commande qui peut être modifiée (voir le paragraphe suivant).

edinburg.mo

Module contenant les déclarations, sous une forme codée, des règles prédéfinies utilisables en syntaxe Edimbourg. On peut s'en passer si on ne souhaite pas utiliser ces règles-là.

Fichiers annexes

exemples

Ce dossier contient un certain nombre de programmes-sources Prolog III.

Le rôle des fichiers du disque 3 et du dossier « sources et objets » est expliqué dans la section « Ajout de règles prédéfinies » de ce chapitre.

Procédure d'installation

Sur disque dur

Copiez les fichiers et dossiers des volumes de distribution dans votre disque de travail, puis rangez les volumes de distribution en un lieu sûr. Pour utiliser Prolog en consommant le moins d'espace disque, vous pouvez ne recopier que les fichiers qualifiés d'indispensables dans la liste précédente. Il vaut mieux copier dans un même dossier les fichiers *Prolog III*, *initial3.psv*, *prolog3.bst*, *prolog3.err*, *prolog3.syn* et *prolog3.prf*.

© PrologIA

Sur Disquettes

Il faut disposer d'au moins deux lecteurs. Mettre sur la disquette système le fichier *initial3.psv* et sur une nouvelle disquette une copie des autres fichiers dits *indispensables*. Vous devez démarrer à partir du dossier contenant *initial3.psv* (en le cliquant par exemple).

2. Utilisation de Prolog III

Activation de Prolog III

Pour commencer une session, ouvrez l'application ***Prolog III*** ou bien un document qui lui est associé (*initial3.psv* par exemple, ou bien un fichier texte ou module crée par une précédente session). Dans tous les cas, le répertoire par défaut (c.a.d. celui pour lequel les fichiers n'auront pas besoin d'avoir un nom complet) est celui dans lequel on a sélectionné le ou les documents à ouvrir (ou à cliquer)

Pour terminer une session, exécutez (sous Prolog III) la requête *quit;* ou bien choisissez l'item **Quit** du menu **Fichier**.

Paramétrisation de l'interprète Prolog III

Si vous souhaitez activer Prolog III en redirigeant les entrées/sorties (ce qui suppose une utilisation non interactive de Prolog III) ou bien en fixant des paramètres d'exécution qu'on modifie rarement, il vous suffit de créer un fichier *prolog3.prf* dans le dossier contenant l'application *Prolog III*. Ce fichier doit contenir une ligne de texte qui possède en réalité la syntaxe et la signification de la commande qui activerait Prolog III dans un environnement de type UNIX,

Seule la première ligne du fichier *prolog3.prf* est prise en compte. Celle-ci doit commencer par *Prolog3*. En voici un exemple relativement complexe (le texte suivant serait en réalité écrit sur une seule ligne) :

```
Prolog3  c10000  u200  b2000  r100000
                        Osortie.log  Ientree.log  q
```

Ces spécifications configurent la session Prolog III de la manière suivante : la pile de copie (*c*) a une taille de 10000 mots, la pile de restauration (*b*) une taille de 2000 mots, l'espace des règles (*r*) dispose de 100000 mots et l'espace auxiliaire des algèbres (*u*) comporte 200 mots. Dans tous les cas il s'agit de mots de quatre octets. En outre, la sortie standard (*O*) est redirigé vers le fichier *"sortie.log"* et les entrées (*I*) sont lues sur le fichier *"entree.log"*. Finalement, l'affichage des messages informatifs autres que les messages d'erreurs est supprimé.

De manière générale, la liste des options a la structure suivante (le signe … signifie qu'un argument doit être présent, collé à l'option) :

```
I… O… Y… c… u… b… r… m… -i… -e… q
```

I*fichier* spécification d'un fichier d'entrée qui, *dès de début de la session Prolog III*, prendra la place du clavier. Ceci implique une utilisation non interactive de Prolog III ; pour cette raison, cette option et sa suivante sont d'un usage exceptionnel.

O*fichier* spécification d'un fichier de sortie qui remplacera l'écran.

Y*fichier* spécification d'un fichier de messages d'erreur. Par défaut, le fichier utilisé est le fichier *prolog3.syn* livré.

c*taille* spécification de la taille, en nombre de mots de 4 octets, de la *pile de copie* (espace pour la construction des structures) Valeur minimum conseillée : 20 000.

b*taille* spécification de la taille, en nombre de mots de 4 octets, de la pile de *restauration* (ou de backtracking) Valeur minimum conseillée : 10 000.

r*taille* spécification de la taille, en nombre de mots de 4 octets, de l'*espace des règles*. Valeur minimum conseillée : 80 000.

u*taille* spécification de la taille, en nombre de mots de 4 octets, de l'espace auxiliaire des algèbres. Valeur minimum conseillée : 200.

© PrologIA

`m`*taille*	spécification de la taille, en nombre de mots de 4 octets, de la pile pour la récupération de mémoire. Valeur minimum conseillée : 1000 .
`-i`*taille*	spécification de la taille, en octets, du dictionnaire interne (liée au nombre de paquets de règles). Valeur minimum conseillée : 35000 .
`-e`*taille*	spécification de la taille, en octets, du dictionnaire externe (contenant les noms de tous les identificateurs). Valeur minimum conseillée : 32000 .
`q`	cette option supprime l'affichage des avertissements et de tous les messages autres que les messages d'erreur.

Interruption d'un programme

Sur Macintosh, l'interruption d'un programme s'obtient :

- soit en choisissant l'item **Stop** du menu **Contrôle**
- soit en pressant simultanément les touches *Commande* et « *;* »

A la suite d'une telle interruption, Prolog III affiche le dialogue :

Les boutons ont la signification suivante :

Continue	:	Continuer, comme s'il n'y avait pas eu d'inter ruption.
Kill	:	Avorter le programme en cours (retour au niveau supérieur de l'interprète Prolog III).

Trace : Bascule de la trace (activation ou désactivation).

Quit : Abandon de la session Prolog III (retour au finder).

Il est possible de taper la première lettre du bouton choisi au lieu de le cliquer. Les touches *ENTER* et *retour chariot* correspondent au bouton par défaut **Continue**. Il est également possible de taper un des chiffres de **0** à **3**, indiquant le niveau de trace (0 si pas de trace, 2 si on désire la trace "ordinaire"). On se référera à la primitive *trace* pour plus de détails.

Il est à noter que la récupération de mémoire n'est pas interruptible.

3. Spécificités de Prolog III Macintosh

Valeurs extrêmes des constantes arithmétiques

1. Les nombres entiers ne sont limités que par la taille de la mémoire de votre ordinateur, et la restriction suivante : le nombre de mots de quatre octets requis pour représenter l'entier en question doit être inférieur à 16 777 215 (cela fait un entier vraiment très grand !).

2. Les nombres flottants sont implémentés avec le type numérique *extended*, conformément à la norme IEEE 754. Codé sur 80 bits, c'est un type plus précis que les *double* classiques sur 64 bits.

© PrologIA

Espaces et tailles par défaut

- Espace principal 50 000 (option **c**)
- Pile de backtracking 10 000 (option **b**)
- Espace secondaire 10 000 (option **u**)
- Espace du code 80 000 (option **r**)
- Pile pour la récupération de mémoire 1 000 (option **m**)
- Dictionnaire interne 36 000 (option **-i**)
- Dictionnaire externe 32 760 (option **-e**)

Toutes ces tailles sont exprimées en nombres de mots de quatre octets, à l'exception des deux dictionnaires où cette taille est donnée en octets.

4. A propos de l'environnement de Prolog III sur Macintosh

Utilisation d'un éditeur hôte

La question de l'utilisation d'un éditeur de textes plus ou moins perfectionné ne se pose pour ainsi dire pas sur le Macintosh. Sauf la fenêtre *console*, toutes les fenêtres de texte ouvertes sous Prolog III comportent un éditeur de textes avec les fonctionnalités standard : copier-coller, rechercher, etc…

La fenêtre *console* est la seule depuis laquelle on peut lancer un programme Prolog III[1] ; en contrepartie, elle ne possède pas les fonctions d'édition des autres fenêtres. Comme un terminal classique, cette fenêtre grossit par ajout de lignes à la suite de celles déjà écrites. La dernière ligne de *console* possède certains privilèges :

- la frappe de la touche *retour chariot* produit la lecture par Prolog III de cette dernière ligne (sans l'éventuel "prompt" que Prolog aurait préalablement affiché) et l'exécution de la commande que cette ligne représente.
- la frappe de la touche *enter* (sur le pavé numérique) produit
  - si aucun texte n'est sélectionné, le même effet que la touche *retour-chariot*
  - si un texte est sélectionné, n'importe où dans la fenêtre console, la recopie de ce texte sur la dernière ligne et la lecture et exécution du tout par Prolog III

Bien entendu, la dernière ligne est éditable et peut être corrigée par les moyens habituels (souris, etc…) aussi longtemps que la frappe de retour-chariot n'a pas terminé la lecture.

5. Ajout de règles prédéfinies

Cette section montre comment ajouter de nouvelles règles prédéfinies à l'ensemble standard, et décrit les procédures d'interface. Ces règles peuvent soit être entièrement écrites en Prolog III (cela ne fait alors intervenir aucun concept nouveau), soit faire référence à des fonctions externes écrites en C ou tout autre langage compatible avec C. Des procédures de communication permettent le passage de paramètres entre les règles Prolog et le programme externe ; ces paramètres doivent être de type entier, réel

[1] La convention par défaut adoptée dans Prolog III n'est pas la même que celle de l'environnement MPW, où toute fenêtre comporte simultanément un éditeur de textes et un interprète de commandes, ce qui a la conséquence d'obliger à réapprendre l'emploi de touches parmi les plus courantes, comme «retour chariot» et «enter». Il est néanmoins possible de se mettre en mode d'édition MPW à l'aide de la commande *stty* .

© PrologIA

ou chaîne de caractères. La fonction externe peut réussir ou, au contraire, provoquer un *backtracking*.

Le module utilisateur (fichier *userrule.c*, livré avec le système Prolog III) sert de relais entre Prolog et les routines de l'utilisateur écrites en C ou dans un autre langage compatible avec C. De cette manière il est possible d'avoir simultanément des règles prédéfinies dans les deux langages.

Marche à suivre

Quand Prolog III est lancé pour la première fois, l'état initial standard de la mémoire de travail (fichier *initial3.psv*) est chargé. Pour ajouter de nouvelles règles prédéfinies écrites dans un autre langage il convient de procéder de la façon suivante :

- choisir un nombre entier spécifique, différent de tous ceux déjà utilisés dans ce but, pour faire le lien entre l'identificateur qui sera le nom de la règle dans Prolog III et celui qui sera le nom de la procédure externe
- associer, dans l'univers Prolog III, ce nombre-relais et le nom de la nouvelle règle ; cela se fait à l'aide de la primitive **set_ident(*nom-de-la-règle*, *numéro*)**. Ultérieurement, il faudra sauvegarder cette déclaration dans une nouvelle version du fichier *initial3.psv*, qui jouera par la suite le rôle d'état initial « enrichi ».
- associer, dans l'univers C, le nombre-relais et l'appel de la fonction qui constitue l'implantation effective de la nouvelle règle. Cela se fait en ajoutant un cas à la table d'aiguillage qui se trouve dans le module `userrule.c`, et en recompilant ce module.
- définir l'implantation externe en C (ou Fortran ou tout autre langage externe compatible) de la règle en question, et la compiler.

Il faut enfin faire l'édition de liens entre le programme externe et les modules objets qui constituent Prolog III, pour produire un exécutable contenant la nouvelle règle.

Appel d'une règle externe

L'interface entre la forme Prolog III d'une règle externe et son implantation effective dans un autre langage est réalisée par une fonction C, qui constitue l'essentiel du fichier *userrule.c*, et qui possède l'en-tête suivant :

```
long UserRule(Ptr_vers_PEV, Num_ident_du_PEV)
    char *Ptr_vers_PEV;
    long int Num_ident_du_PEV;
```

Le listing du fichier *userrule.c* se trouve à la fin de ce chapitre. Vous devez savoir que, pour exécuter une règle externe, Prolog III appelle cette fonction avec pour arguments les valeurs suivantes :

Ptr_vers_PEV est l'adresse du terme Prolog III correspondant à l'appel de la règle externe (c'est-à-dire l'adresse du but à effacer). Il s'agit donc un arbre, dont l'étiquette est le nom de la règle externe et dont les fils sont les arguments de l'appel.

Num_ident_du_PEV est un nombre entier identifiant la règle externe en question.

UserRule(Ptr_vers_PEV, Num_ident_du_PEV)

La valeur retournée par la fonction *UserRule* détermine si l'exécution de la règle externe a réussi ou échoué. Si *UserRule* renvoie une valeur non nulle (par exemple, la constante TRUE) alors Prolog III considère que, pour ce qui concerne le programme externe, l'effacement a réussi. Si au contraire *UserRule* renvoie zéro (FALSE) alors l'exécution de la règle externe est considérée comme se soldant par un échec, ce qui produira un backtracking.

L'essentiel de la fonction *UserRule* est constitué par une instruction d'aiguillage (*switch*) portant sur la valeur *Num_ident_du_PEV*. Pour ajouter une nouvelle règle externe à Prolog III, il vous suffit d'ajouter un « cas » à cet aiguillage, en correspondance avec le numéro choisi pour représenter la règle. En général, pour une nouvelle règle externe vous ajouterez les instructions C suivantes :

- d'abord, un certain nombre d'appels de la fonction *GetArg*, pour obtenir des pointeurs vers les arguments de l'appel de la règle externe

© PrologIA

- en correspondance avec les précédents, un certain nombre d'appels des fonctions « *Convert_MachinP_to_TrucC* », pour amener aux conventions de C le codage des valeurs des arguments
- ensuite, l'appel d'une fonction C, écrite par vous, qui constitue l'implantation effective de la règle externe
- au retour de cette fonction, un certain nombre d'appels de fonctions de conversion « *Convert_TrucC_to_MachinP* », afin de coder les résultats selon les conventions internes à Prolog III
- finalement, des appels de la fonction *PutArg*, pour donner ces résultats pour valeurs à certains des arguments de la règle d'appel.
- sans oublier de préciser la valeur (vraie ou fausse) qui doit être retournée par la fonction *UserRule* à titre de compte rendu (succès ou échec) de l'exécution de la règle externe.

Vous trouverez dans les paragraphes suivants plusieurs exemples d'implantation de règles externes.

Procédures de transfert de données

Seuls des entiers, des réels et des chaînes de caractères peuvent actuellement être échangés entre un programme Prolog III et des routines externes écrites en C ou un autre langage. Deux jeux distincts de procédures de transfert de données sont fournis, correspondant à deux manières légèrement différentes d'organiser le travail :

- un premier système, formé des procédures C :
  GetArity
  GetArg
  PutArg
  Convert_StringC_to_StringP
  Convert_LongIntC_to_IntP
  Convert_ExtendedC_to_NumP
  Convert_DoubleC_to_NumP
  Convert_StringP_to_StringC
  Convert_NumP_to_ExtendedC
  Convert_NumP_to_DoubleC
  Convert_IntP_to_LongIntC
- un deuxième système de transfert de données, compatible avec celui de Prolog II+, formé des procédures C :
  get_integer
  get_real
  get_string
  put_integer
  put_real
  put_string

Premier protocole de transfert de données.

Le premier système se compose d'un jeu de fonctions générales et de fonctions de conversion. Il faut donc procéder en deux temps.
Pour récupérer une donnée à partir d'une structure Prolog, récupérer l'argument correspondant (fonction *GetArg()*) puis le convertir.
Pour fournir une donnée à une structure Prolog, convertir cette donnée en un terme Prolog puis instancier l'argument correspondant (fonction *PutArg()*).

Fonctions générales :

```
long GetArity(term)
     adr term;

adr GetArg(no_arg,term)
     long int no_arg;
     adr term;
```


© PrologIA

long PutArg(no_arg,term,value)
long int no_arg;
adr term;
adr value;

term désigne l'adresse[1] d'un terme, dont on s'intéresse à l'arité (*GetArity*) ou à un fils (*GetArg*, *PutArg*). En général, *term* pointera l'appel de la règle externe, obtenue à travers l'argument *Ptr_vers_PEV* de l'appel de *UserRule*. Notez cependant que cela n'est pas obligatoire, et qu'on peut utiliser *GetArity*, *GetArg* et même *PutArg* pour manipuler des termes autres que celui de l'appel de la règle externe.

no_arg est un entier qui, lorsqu'il est supérieur ou égal à 1, représente le rang du fils du terme précédent auquel on s'intéresse, que ce soit pour obtenir sa valeur (*GetArg*) ou pour la définir (*PutArg*).

value donne l'adresse de la représentation interne d'un terme Prolog III, provenant généralement d'un appel d'une des fonctions de conversion : *Convert_StringC_to_StringP*, *Convert_Long-IntC_to_IntP*, *Convert_DoubleC_to_NumP.*

GetArity(term)
Retourne l'arité, ou nombre d'arguments, du terme pointé par *term*.

GetArg(no_arg, term)
Retourne un pointeur vers le terme qui est le fils de rang *no_arg* du terme pointé par *term*. Il se produit une erreur si cet argument n'existe pas, indiquée par le fait que la fonction retourne *NULL*.
En outre, *GetArg(0, term)* retourne un pointeur vers l'étiquette de *term*.

PutArg(no_arg, term, value)
Unifie le terme pointé par *value* avec le fils de rang *no_arg* du terme pointé par *term*. Il se produit une erreur si un tel fils n'existe pas ou si une telle unification n'est pas possible.
En outre,*PutArg(0,term,value)* unifie le terme pointé par *value* avec l'étiquette de *term*.

---

1 Le type *adr* représente l' « adresse la plus générale » i.e. le type « `char *` »

Fonctions de conversion :

```
adr Convert_StringC_to_StringP(buf)
      char* buf;

long Convert_StringP_to_StringC(t,buf,size_buf)
      adr t;
      char* buf;
      long int size_buf;

adr Convert_LongIntC_to_IntP(n)
      long int n;

int Convert_IntP_to_LongIntC(t,i)
      adr t;
      long int* i;

adr Convert_DoubleC_to_NumP(f)
      double f;

int Convert_NumP_to_DoubleC(t,f)
      adr t;
      double* f;

adr Convert_ExtendedC_to_NumP(f)
      double f;

int Convert_NumP_to_ExtendedC(t,f)
      adr t;
      double* f;
```

Convert_StringC_to_StringP(buf)

Transforme la chaîne C (terminée par zéro) pointée par *buf* en une chaîne Prolog III et retourne l'adresse du terme ainsi construit, ou *NULL* si la transformation n'a pas pu être faite.

© PrologIA

Convert_StringP_to_StringC(t,buf,size_buf)
Transformation réciproque de la précédente : *t* doit être l'adresse d'un terme Prolog III de type chaîne de caractères ; *buf* est l'adresse d'un espace ayant pour taille au moins *size_buf* caractères. Cette fonction transforme la chaîne Prolog pointée par *t* en une chaîne C terminée par zéro, rangée à partir de l'adresse *buf*. Si la taille de la chaîne Prolog dépasse la valeur *size_buf*, alors la chaîne est tronquée.

Convert_LongIntC_to_IntP(n)
Transforme l'entier long *n* en un nombre Prolog III et retourne l'adresse du terme ainsi construit, ou *NULL* si la transformation n'a pas pu être faite.

Convert_IntP_to_LongIntC(t,i)
Transformation réciproque de la précédente ; *t* doit être l'adresse d'un terme Prolog III de type nombre (entier). Cette fonction dépose dans la variable entière pointée par *i* la valeur de ce nombre. Une erreur se produit si la transformation ne peut pas être faite. La fonction retourne 0 (*FALSE*) en cas d'erreur, une valeur non nulle (*TRUE*) autrement.

Convert_ExtendedC_to_NumP(f)
Convert_DoubleC_to_NumP(f)
Transforme le réel de « grande précision » *f* en un nombre Prolog III et retourne l'adresse du terme ainsi construit, ou *NULL* si la transformation n'a pas pu être faite.
Dans le premier cas, il s'agit du type *extended* (flottant sur 80 bits) associé à la bibliothèque SANE ; dans le second, il s'agit du type *double* standard. Puisque le C du Macintosh transforme tous les flottants en *extended* pour l'évaluation des expressions arithmétiques, si on ne regarde que l'efficacité ou la précision du programme on a intérêt à travailler avec des *extended*. Si au contraire on tient beaucoup à la portabilité, alors on préférera le type *double*.

Convert_NumP_to_ExtendedC(t,f)
Convert_NumP_to_DoubleC(t,f)
Transformation réciproque de la précédente ; *t* doit être l'adresse d'un terme Prolog III de type nombre. Cette fonction dépose dans la variable réelle de « grande précision » pointée par *f* la valeur de ce nombre. Une erreur se produit si la transformation ne peut pas être faite. La fonction retourne 0 (*FALSE*) en cas d'erreur, une valeur non nulle (*TRUE*) autrement.

Second protocole de transfert de données.

Le deuxième système offre, pour chacun des types de données permis (entiers, réels, chaînes), une fonction associée pour transférer cette donnée depuis Prolog vers le programme externe, et une autre pour la transférer depuis le programme externe vers Prolog. Pour chaque argument de la règle externe (c'est-à-dire du but à effacer), la fonction adéquate doit être appelée avec comme paramètre le rang de cet argument.

Transfert de données simples de Prolog vers C.

Ces fonctions sont appelées par le programme externe pour obtenir les valeurs effectives des arguments de la règle externe. Une fonction spécifique est associée à chaque type de donnée.

Si le type de l'argument effectif n'est pas celui attendu, la fonction de communication renvoie une erreur et force le programme à *backtracker*. Attention : il est alors impératif de sortir immédiatement de la fonction externe.

Voici les fonctions disponibles pour les types simples :

```
get_integer(no_arg,value,err)
    int no_arg;
    long *value;
    int *err
```


© PrologIA

get_real(no_arg,value,err)
int no_arg;
*float *value;*
*int *err;*

get_string(no_arg,value,err)
int no_arg;
*char *value;*
*int *err;*

no_arg est un entier donnant le rang de l'argument choisi dans l'appel de la règle externe. Le premier argument de la règle externe a le rang 1, le second a le rang 2 et ainsi de suite. Si la valeur effective de *no_arg* ne correspond pas à un argument effectif dans la règle externe, alors il se produit une erreur.

value est l'adresse de la variable qui est l'argument de rang *no_arg* dans la règle externe. Si son type ne correspond pas à celui requis par la fonction, un backtracking est automatiquement généré. Pour les paramètres de type chaîne de caractères, value doit être l'adresse d'un tableau, de taille suffisante pour contenir la chaîne, défini dans le programme externe.

err est une variable positionnée à 1 si une erreur s'est produite, ou si un backtracking a été généré. Le programme externe n'est pas autorisé à récupérer ce type d'erreur. Pour un comportement correct du système de gestion des erreurs, on doit immédiatement sortir du programme externe si *err* est non nulle.

Ces fonctions retournent une valeur qui est l'opposée, en tant que valeur booléenne, de l'argument *err*.

get_integer(no_arg,value,err)
Vérifie que l'argument de rang *no_arg* est un entier (pouvant tenir dans un *int* C) et le transfère dans la variable (ou la zone mémoire) dont *value* est l'adresse.

get_real(no_arg,value,err)
Vérifie que l'argument de rang *no_arg* est un flottant et le transfère dans la variable (ou la zone mémoire) dont *value* est l'adresse.

get_string(no_arg,value,err)
Copie la chaîne de caractères originale (depuis la mémoire de travail de Prolog) dans une variable (ou zone mémoire) définie dans le programme externe. Un caractère nul indique la fin de la chaîne.
La zone d'accueil doit être de taille suffisante pour contenir les caractères et le caractère nul.

Transfert de données simples de C vers Prolog.

Ces fonctions sont appelées par le programme externe pour unifier une valeur avec un argument de la règle externe. Si l'unification échoue, un backtracking est automatiquement généré par la fonction de communication. Voici les fonctions disponibles :

put_integer(no_arg,value,err)
```
int no_arg;
long value;
int *err
```

put_real(no_arg,value,err)
```
int no_arg;
float value;
int *err;
```

put_string(no_arg,value,err)
```
int no_arg;
char *value;
int *err;
```

no_arg est un entier donnant le rang de l'argument dans l'appel de la règle externe. Si la valeur effective de *no_arg* ne correspond pas à un argument effectif dans la règle externe, alors il se produit une erreur.

value est la valeur devant être unifiée sur l'argument de rang *no_arg* dans la règle externe.

© PrologIA

Pour les paramètres de type chaîne de caractères, *value* est l'adresse d'une chaîne de caractères terminée par zéro, définie dans le programme externe. *put_string* copie cette chaîne de caractères dans la mémoire de travail de Prolog avant d'unifier la valeur avec l'argument de la règle externe.

err est une variable positionnée à 1 si une erreur s'est produite, ou si un backtracking a été généré. Le programme externe n'est pas autorisé à récupérer ce type d'erreur. Pour un comportement correct du système de gestion des erreurs, on doit immédiatement sortir du programme externe si *err* est non nulle.

put_integer(no_arg,value,err)
Unifie l'argument de rang *no_arg* avec l'entier contenu dans *value.*

put_real(no_arg,value,err)
Unifie l'argument de rang *no_arg* avec le flottant contenu dans *value.*

put_string(no_arg,value,err)
Copie la chaîne de caractères pointée par value dans la mémoire de travail de Prolog et l'unifie avec l'argument de rang *no_arg* .
Noter qu'un caractère nul doit indiquer la fin de la chaîne originale.

Ces trois fonctions retournent une valeur qui est l'opposée, en tant que valeur booléenne, de l'argument *err*. Ainsi, on peut considérer que la fonction est évaluée à TRUE lorsqu'il n'y a pas eu d'erreur, à FALSE lorsqu'une erreur s'est produite. Cela rend équivalentes les deux formes d'appel suivantes :

```
...
put_integer(n,v,err);
if (*err)
    return;
...
```

et

```
...
if ( ! put_integer(n,v,err))
    return;
...
```

Un exemple complet

Voici un exemple complet, décrivant l'ensemble des opérations à exécuter pour créer une nouvelle règle prédéfinie implantée par un programme externe.

Supposons que vous programmiez en C et que vous vouliez ajouter la règle prédéfinie *roots(a, b, c, x, y)* qui calcule les deux racines réelles x et y de l'équation : $ax^2 + bx + c = 0$ si elles existent et qui provoque un backtracking sinon.

S'agissant du Macintosh, vous devez posséder l'atelier de programmation MPW. Nous supposerons que c'est le cas, et que vous connaissez le mode d'emploi de ce logiciel. Etape par étape, la marche à suivre est la suivante :

1. Créer le fichier *myproc.c* suivant :

```
#include "macuser.h"
#include <SANE.h>

real_roots(a, b, c, x, y)
    extended a, b, c, *x, *y;
    {
    extended d;

    if (a == 0 || (d = b * b - 4 * a * c) < 0)
        return 0;
    else
        {
        *x = (-b + sqrt(d)) / (2 * a);
        *y = -b / a - *x;
        return 1;
        }
    }
```

2. Compiler le fichier *myproc.c* :

```
C  -s  _MYSEGT_  myproc.c
```

3. Modifier la fonction *UserRule* du fichier *user_rule.c* en ajoutant l'appel du nouveau programme externe. Par exemple, en donnant le numéro 504 à la nouvelle règle :

```
long int UserRule(Ptr_vers_PEV, Num_ident_du_PEV)
    adr Ptr_vers_PEV;
    long int Num_ident_du_PEV;
```


© PrologIA

```
    {
   ...
    switch (Num_ident_du_PEV)
        {
      ...
        case 504:
            {
            adr t;
            extended a, b, c, x, y;

            return ( ! ((t = GetArg(1, Ptr_vers_PEV))
                   && Convert_NumP_to_ExtendedC(t, &a)
                   && (t = GetArg(2, Ptr_vers_PEV))
                   && Convert_NumP_to_ExtendedC(t, &b)
                   && (t = GetArg(3, Ptr_vers_PEV))
                   && Convert_NumP_to_ExtendedC(t, &c)

                   && real_roots(a, b, c, &x, &y)

                   && (t = Convert_ExtendedC_to_NumP(x))
                   && PutArg(4, Ptr_vers_PEV, t)
                   && (t = Convert_ExtendedC_to_NumP(y))
                   && PutArg(5, Ptr_vers_PEV, t)))
               ? FALSE : TRUE;
            }
      ...
        }
    }
```

4. Modifier la commande *Link* dans le fichier *prolink3* pour tenir compte du nouveau fichier (faites attention à ne pas laisser de blanc après le caractère « ∂ ») :

```
...
 Link -p -d -c 'PRO4'  -t APPL -o 'Prolog III'.new ∂
    -b -sn edit=Main ∂
    myproc.c.o  ∂
    prolog3.o userRule.c.o promain.c.o ∂
    {Libraries}RunTime.o {Libraries}interface.o ∂
    {CLibraries}CRuntime.o {CLibraries}StdCLib.o ∂
    {CLibraries}CInterface.o {CLibraries}Math.o ∂
    {CLibraries}CSANELib.o
 Setfile -a B 'Prolog III'.new
...
```

5. Refaire la compilation de *userrule.c* et l'édition de liens de *Prolog III* :

```
make -f prolink3 > tmp ; tmp
```

6. Lancer une session Prolog III utilisant le nouvel exécutable et l'ancien état initial, définir le nom de la nouvelle primitive et l'ajouter à la partie fermée de la famille "sys" :

```
PROLOG III, v1.2 Decembre 1990 (C) PrologIA 1989-90

> set_ident(sys:roots, 504);
{}
> add_implicit("sys","roots");
{}
>
```

7. Essayer la nouvelle règle :

```
> roots(1,-5,6,x,y);
{x=2, y=3}
> roots(1,4,8,x,y);
>
```

8. Sortir de Prolog III en sauvant l'état courant. Celui-ci devient l'état initial pour les futures sessions Prolog III :

```
> exit("initial3.psv");
```


© PrologIA

6. Appeler Prolog III depuis le langage C

Introduction

Appeler Prolog III depuis C se fait de façon simple puisque trois fonctions C autorisent ce mécanisme.

De par la nécessité d'allouer et d'initialiser ses propres donnés, Prolog III ne pourra être appelé qu'une fois la fonction *Pro3Initial* exécutée. Les appels à Prolog III se font ensuite par l'usage de la fonction générique *Pro3Goal*, qui envoie à Prolog III une commande sous la forme d'une chaîne de caractères. Dans le cas où on n'aurait plus besoin de Prolog III dans la suite de l'exécution, la fonction *Pro3Final* peut être appelée; celle-ci permet notamment de récupérer la place mémoire occupée par la session Prolog III. On ne pourra plus à partir de ce moment-là appeler Prolog III à moins de recommencer le processus en utilisant à nouveau *Pro3Initial*, etc….

Description des fonctions

Initialisation

```
int Pro3Initial(argc, argv, autoprolog)
int argc, autoprolog;
char *argv[];
```

argc, *argv* ont la même sémantique que les arguments de la fonction *main()* de tout programme C. Ils servent à passer des options à Prolog III. Dans le cas où on ne souhaite pas utiliser cette facilité, on peut passer *0* pour *argc* et *NULL* pour *argv*. Rappelons que, lorsqu'il existe un fichier nommé *prolog3.prf* dans le répertoire courant (ou dans celui contenu dans la variable d'environnement *PrologDir3* (si elle est définie)), sa première ligne est interprété comme une ligne de commande unix et les options y sont lues. Dans ce cas, les arguments *argc* et *argv* sont ignorés.

autoprolog informe Prolog III qu'il est le programme principal ou qu'il est seulement appelé de temps à autre. Les valeurs possibles à donner sont :

PROMAIN si Prolog III est le programme principal

PROCALL si Prolog III est utilisé comme module externe

Pro3Initial(argc, argv, autoprolog)

Cette fonction alloue puis initialise Prolog III. Cette initialisation comprend le démarrage de Prolog III . Il peut se passer un assez long moment avant qu'il ne rende la main (selon la taille de l'état qu'il doit charger).
Cette fonction retourne un statut d'erreur, lequel est ERR3_NOERR lorsqu'il n'y a pas d'erreur et que Prolog III a pu allouer, charger, initialiser son espace, démarrer, ….

Toutes les constantes *PROMAIN*, *PROCALL*, *ERR3_NOERR*, etc, sont définies dans le fichier d'entête *macuser.h* qu'il faut donc inclure.

Appel

```
int Pro3Goal(g)
char *g;
```

Pro3Goal(g)

On appelle Prolog III à travers cette unique fonction. Son argument *g* est une chaîne de caractères C (terminée par '\0') qui contient une requête Prolog III. On notera qu'avec cette version le passage de données ne se fait que de C vers Prolog III, celui-ci donnant ses résultats sur la sortie courante de Prolog III (laquelle peut-être modifiée).

Cette fonction retourne également un statut d'erreur. Quand il n'y a pas eu d'erreur, *ERR3_NOERR* est retourné. Sinon est retourné :

- un nombre positif correspondant à une erreur pendant l'exécution de Prolog III.
- *ERR3_NOINI* Si Prolog III n'est pas initialisé.

© PrologIA

- *ERR3_ERRANA* S'il y a une faute de syntaxe dans la commande.

Par exemple, l'appel :

```
err = Pro3Goal("enum(x), {0<x<4};");
```

produira à l'exécution les affichages suivants :

```
{x = 1}
{x = 2}
{x = 3}
```

et la variable *err* vaudra *ERR3_NOERR.*

Terminaison

```
int Pro3Final()
```

Pro3Final()
Récupère les espaces alloués par Prolog III. L'appel de cette fonction ne se justifie guère que pour récupérer une importante place mémoire. C'est ici également que les fichiers ouverts par Prolog III sont fermés. Les retours possibles de cette fonction sont les deux statuts d'erreur suivants :

- *ERR3_NOERR* pas d'erreurs
- *ERR3_NOINI* Prolog III n'était pas initialisé

Un petit exemple

Voici comme illustration un petit exemple résumant ce qui a été dit plus haut. Dans cet exemple, Prolog III est supposé être utilisé "intensément" en deux endroits du programme, et on souhaite récupérer l'espace Prolog III entre les deux.

```
/* fichier monprog.c */
#include "macuser.h"
main()
    {
    int err;
    ...travail....
```

```
err = Pro3Initial(0, NULL, PROCALL);
if ( err != ERR3_NOERR ) {
   printf("Pas pu initialiser Prolog III\n");
   exit(1);
   }
...travail....
err = Pro3Goal("out(c), {c=1/2};");
...travail....
err = Pro3Goal("out(c) line, {c=1/4};");
...travail....
Pro3Final();
...travail....
...travail....
...travail....
err = Pro3Initial(0, NULL, PROCALL);
if ( err != ERR3_NOERR ) {
   printf("Impossible d'initialiser Prolog\n");
   exit(1);
   }
...travail....
err = Pro3Goal("out(c), {c=1};");
...travail....
err = Pro3Goal("outl(c) fail, {c=2};");
...travail....
Pro3Final();
...travail....
}
```

En supposant que les lignes *...travail....* sont vides, compilons ce fichier en le liant à Prolog III. On va produire l'exécutable nommé *monprog* :

```
c -o monprog.c.o monprog.c
```

Il faut pour l'édition de lien modifier le fichier *prolink3* en remplaçant *promain.c.o* par *monProg.c.o* et en renommant après le *link* le fichier *'Prolog III'.new* en *monProg* :

```
...
Link -p -d -c 'PRO4'  -t APPL -o 'Prolog III'.new ∂
    -b -sn edit=Main ∂
    monProg.c.o  ∂
    prolog3.o userRule.c.o ∂
    {Libraries}RunTime.o {Libraries}interface.o ∂
    {CLibraries}CRuntime.o {CLibraries}StdCLib.o ∂
    {CLibraries}CInterface.o {CLibraries}Math.o ∂
```

```
    {CLibraries}CSANELib.o
Setfile -a B 'Prolog III'.new
Move 'Prolog III'.new   monProg
...
```

On refait la compilation de *userrule.c* et l'édition de liens de *monProg* :

```
make -f prolink3 > tmp ; tmp
```

Lançons-le et voyons les résultats affichés dans la console :

```
PROLOG III, v1.2 Decembre 1990 (C) PrologIA 1989-90

1/2{c = 1/2}
1/4
{c = 1/4}


PROLOG III, v1.2 Decembre 1990 (C) PrologIA 1989-90

1{c = 1}
2
```

Listings

Fichier *macuser.h* :

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

#define ERR3_NOERR    0
#define ERR3_NOINI   -2
#define ERR3_ERRANA  -3
#define ERR3_DEJAINI -4
#define ERR3_ABANDON -5

#define PROMAIN 1
#define PROCALL 0

typedef char* adr;

extern FILE *entree, *sortie, *sorerr;


extern  adr
    GetArg(/* long int n,  adr t */),
    Convert_StringC_to_StringP(/* char* buf */),
```

```
    Convert_LongIntC_to_IntP(/* long int  n */),
    Convert_DoubleC_to_NumP(/*  double d */);

extern long int
   GetArity(/* adr terme */)  ,
   PutArg(/* long int n,  adr T,  adr t */),
   Convert_StringP_to_StringC(/*adr t, char* buf,
                    long int size_buf */),
   Convert_NumP_to_DoubleC(/* adr t, double *f */),
   Convert_IntP_to_LongIntC(/* adr t, long int* i */);

int Pro3Initial(/* int argc, char** argv, int autoprolog */);
int Pro3Goal(/* char *g */);
int Pro3Final();

#ifdef mpw
extern  adr
   Convert_ExtendedC_to_NumP(/* extended  d */);
long int
   Convert_NumP_to_ExtendedC(/* adr t, extended *f */);
#endif
```


© PrologIA

Fichier *userrule.c* :

```
#include "macuser.h"

extern adr TheCurrentPev;

long int UserRule(Ptr_vers_PEV, Num_ident_du_PEV)
    adr Ptr_vers_PEV;
    long int Num_ident_du_PEV;
    {
    int err = 0, err_nb = 0;

    TheCurrentPev = Ptr_vers_PEV;

    switch (Num_ident_du_PEV)   /* Utilisez des numeros */
       {                        /* superieurs  a 500    */

            /* inserez ici vos propres regles */

       default:
            ProPrintf(sorerr,
                          "User Rule (%ld) Not Defined.\n",
                          Num_ident_du_PEV);
            return FALSE;
       }
    }
```

© PrologIA

Index

! (coupure) 164
, 418
-> 417
| (coupure) 164
< 411
<> (tuple vide) 27
= 412
=.. 410
=:= 415
=< 411
== 413
=\= 416
> 411
>= 411
[ ] (liste vide) 33
[ ] (symbole fonctionnel) 83
[] 420
\== 414

A

abandon de l'exécution d'un but (block_exit) 166
abolish 377
abs 172; 369
activate_button 515
add 170; 368
add_implicit 183; 202
add_stdmenu 497
addition 30; 98
affectation 47
affectation (assign) 173
affectation booléenne 125
ajout d'une règle (assert) 186
appels externes à Prolog III 1025
"apple" 497
arbres 20; 70
arbres factuels 71
arbres infinis 75
arg 81; 203
arg2 84; 204
arg3 80; 205
arithmétique 14
assert 186; 206; 378
assert'' 187; 208
asserta 186; 207; 380
assertz 187; 208; 378
assign 173; 174; 209
atan 172; 370
atom 381
atomic 382

B

bitmap de rafraîchissement 468
block 166; 212
block_exit 166; 212
boîte à cocher 500
bool 129; 214
booléens 16; 118
  syntaxe 428
bouclage 162
bound 149; 167; 215
bound_conc 80; 216
bound_mult 105; 156; 217
bound_size 80; 151; 218
bound_tree 219
bound_tree' 220

bouton 485; 501
bouton de terminaison 502
buts 60
syntaxe 440

C

call 383
caractères 25
caractère d'échappement (\) 430
caractère désigné 430
espace 428
remarques 441
syntaxe 431
chaînes de caractères 13; 27; 71; 85
syntaxe 431
changement de signe 30
char 221
char_code 222
chargement d'un programme (input) 8
check_item 496
clause 384
clear_menubar 497
clear_window 472
clic 480; 486; 490; 491; 504; 510
close_context_dictionary 183; 223
close_input 9; 195; 224
close_output 11; 197; 225
cochage d'un item 496
command_menu 496
commentaire 442; 459
comparaisons numériques 52
conc3 80; 154; 226
conc_string 85; 227
concaténation 12; 76
retardée 154
configuration de Prolog III 1006
connu (terme) 148
console 6; 464; 1009
constantes 23
booléennes 119
numériques 97
syntaxe 428
constructeur
d'arbres 72
de tuples 72
général d'arbres 72
consult 385
contexte courant 179
contexte de lecture-écriture 24; 179
contraintes 54; 438
booléennes 16; 120
exemples 122
forme normale 123
simplification 127
contrainte de taille 76
numériques 101
exemples 101
forme normale 102
retardées 156
retardées 151
syntaxe 439
contraintes de type
!bool 53; 120
!boolt 53; 120
!char 53
!chart 53
!id 53
!idt 53
!num 53; 100
!numt 53; 100
!tuple 53
contraintes numériques 14
"control" 497
contrôle 162
Convert_DoubleC_to_NumP 1014; 1017
Convert_ExtendedC_to_NumP 1014; 1017
Convert_IntP_to_LongIntC 1014; 1017
Convert_LongIntC_to_IntP 1014; 1017
Convert_NumP_to_DoubleC 1014; 1018
Convert_NumP_to_ExtendedC 1014; 1018
Convert_StringC_to_StringP 1014; 1016
Convert_StringP_to_StringC 1014; 1017
coordonnées horizontales 473
coordonnées verticales 473
cos 172; 370
coupure 164
cpu_time 198; 228
crayon 474; 481; 482
create_button 515
create_window 471

© PrologIA

current_context 182; 229
current_predicate 187; 230
cut 231
cut (coupure) 164

D

def_array 174; 232
default 167; 234
définition de contexte (set_context) 180
deactivate_button 515
dialogue de saisie de nom de fichier 516
dialogue de sélection d'un fichier 516
dictionary 184; 235
dictionnaire externe (taille) 1009
dictionnaire interne (taille) 1009
dif 237
"DISTANCE" 468
div 106; 171; 238; 368
division 31; 98
dot 239

E

echo 197; 240; 341
edinburgh 241
edinburg.mo 1004
"EDIT" 467
"edit" 497
edit 242
éditeur de textes hôte 1009
égalité 50
enable_menu 496
end_module 185; 243
ensemble de contraintes
booléennes 122
enter 1010
entrée-sorties 192
entree.log 1006
enum 104; 244
epsilon_float 342
eq 246
eql 171; 369
escape 431
espace du code (taille) 1009
espace principal (taille) 1009
espace secondaire (taille) 1009
espacement horizontal 505
espacement vertical 504
et 29; 119
exécution d'un programme 64
exemples
arbres 71
caractères 40
chaînes 41
constructeurs 73
contraintes 56
contraintes booléennes 122
contraintes numériques 101
contraintes sur les arbres 74
contraintes sur les chaînes 85
contraintes sur les tuples 78
entiers 39
expressions booléennes 43; 120
expressions numériques 42; 99
flottants 40
freeze 149
identificateurs 38
listes 84
modules 186
règles 61
requêtes 61
termes 38; 45
utilisation de la coupure (!) 164
exemples de programmes
additionneur binaire 135
au plus un vrai 131
calcul bancaire 108
calcul des feuilles 87
ChatsOiseaux 15
communications Prolog III/C 1022
crible d'Eratosthène 90
element_de 13
K éléments vrais 133
naive reverse 82
puzzle logique 143
quick sort 87
remplissage d'un rectangle 110
RepasLeger 9
SEND + MORE = MONEY 109
suite périodique 89

un ou sur 2 variables 129
un ou sur une liste 130
exit 198; 247
exp 172; 370
expressions
booléennes 43; 119; 437
numériques 42; 97; 436
maximum 105
minimum 105
prédicat val 168

F

fail 249
faits vrais 62
famille 178
partie fermée 178
Fdecimal 108
fenêtre graphique courante 473
feuille 21
Fexact 108
Ffloat 108
"file" 497
file_window 466
fin de session (quit) 11
"find" 497
find_pattern 85; 251
findall 250
Fint+fract 108
Fnormal 108
fonctions externes C 1010
fonctions évaluables 170
"FONT" 469; 471
fonte 472; 482; 483; 484
"FONTSIZE" 468; 471
format_decimal 342
format_float 342
format_out_num 108; 341
Fdecimal 341
Fexact 341
Ffloat 341
Fint+fract 341
Fnormal 341
formats d'entrée-sorties 108
forme clausale 124
forme normale
contraintes booléennes 123
contraintes numériques 102
forme normale conjonctive 123
formfeed 431
free 149; 168; 252
free_label 253
freeze 157; 254
front_window 466
functor 386

G

garbage_collection 255
gcd 256
get 387
get0 388
get_config 257
get_configuration 258
get_integer 1014; 1018; 1019
get_key 487
get_real 1014; 1019; 1020
get_screen 466
get_string 1014; 1019; 1020
get_window 466
GetArg 1014; 1015
GetArity 1014; 1015
getenv 259
gr_arc 476
gr_arc' 477
gr_button_hit 486
gr_choosecolor 482
gr_click 479; 480
gr_color 481
gr_color2 481
gr_color3 481
gr_dialog 498
gr_draw_buttons 485
gr_editf 486
gr_erase 474
gr_font 484
gr_getmouse 480
gr_icon 478
gr_line 474
gr_lineto 474

© PrologIA

gr_list 489
gr_load 516
gr_mode 482
gr_move 474
gr_moveto 474
gr_pen 482
gr_penloc 474
gr_polygon 476
gr_popupItem 491
gr_print 492
gr_rect 475
gr_setorigin 474
gr_stringwidth 482
gr_text 483
gr_window 472
gr_window_is 472
"GRAPHICS" 467
groupe 500
gtty 489

I

ident 260
identificateurs 24; 433; 444
  abrégés 177
    syntaxe 177; 434; 444
  complets 177
  séparateur 177
  syntaxe 177; 434; 444
if 172; 369
implique (opération) 29; 119
implique (relation) 52; 120
in_char 193; 261
in_char' 262
in_ident 263
in_integer 264
in_real 265
in_sentence 194; 266
in_sentence' 267
in_string 268
in_term 269
inégalité 50; 101
inéquations numériques 104
inf 171; 369
infe 171; 369
inférieur ou égal 101
inférieur strict 101
initial3.psv 1004
inl 270
input 8; 194; 271; 516
input_is 272
insert 188; 189; 273
installation 1002
installation de Prolog III 1002
integer 106; 275
interruption 167; 1008
  d'un programme 1007
is 389
is_bool 129; 276
is_char 277
is_ident 278
is_leaf 279
is_num 106; 280
is_tuple 81; 281
is_univ 282
item coché 494

J

jeu de caractères 427

K

kill_button 515
kill_module 12; 191; 283
kill_window 472
kit de distribution 1002
known 148; 284
known_part 81; 152; 285

L

lcm 286
ligne de texte éditable 501
ligne de texte non éditable 500
line 197; 287
linefeed 431
list 7; 9; 11; 189; 288
list_string 86; 293
list_tuple 84; 294

liste vide 84
syntaxe 429
listes 33; 83
listing 390
littéral 124
ln 172; 370
load_graphic 464

M

MacDraw 516
machine Prolog III 65; 162
MacPaint 516
max_value 105; 295
maximum d'une expression numérique 105
member 391
menu fugitif 493
menu hiérarchique 493
menu pop-up 493
min_value 105; 296
minimum d'une expression numérique 105
mod 106; 171; 297; 369
"MODAL" 467
modes de transfert 483
modifications de la taille des piles 1006
module 24; 178; 298
objet 186
source 185
utilisateur 1011
module(directive) 185
modules
exemples 186
moins unaire 98
motif 482
"MPW" 489
mul 170; 368
mult 105; 157; 158; 299
multiplication 31; 98

N

naive_reverse 82
name 392
new 300
new_window 467
newline 431
next_char 301
next_char' 302
nil 33
nl 393
"NLINES" 468
no_echo 197; 303
no_trace 304
nombres 14
entiers 26
syntaxe 428
flottants 26; 94
syntaxe 429
fractionnaires 94
rationnels 94
non 29; 119
non (prédicat) 166
nonvar 394
not 305
num 106; 306
number 395
numden 307

O

omission de l'opérateur de multiplication 99
op 396
opérateurs 445
booléens
équivalent 119
et 119
implique 119
non 119
ou 119
Edimbourg 459
numériques 98
opérations 28
arithmétiques 30; 98
booléennes 29
concaténation de tuples 32; 76
construction d'arbre 31; 71
construction de tuple 32; 71
construction générale d'arbre 32; 71
option fermeture 473

© PrologIA

ou 29; 119
out 195; 308
outc 196; 309
outl 195; 310
outm 196; 311
outml 7; 196; 312
output 11; 197; 313; 516
output_is 314

P

paire pointée 33; 83
paramètres graphiques
:button 501
:button2 501
:buttonD 502
:cb 500
:ccol 504
:clipRect 476
:col 504
:crow 504
:editf 501
:erase 475; 476; 477
:eraseOval 475
:eraseRect 475
:eraseRoundRect 475
:frame 475; 476
:frameOval 475
:frameRect 475
:frameRoundRect 475
:glist 502
:group 505
:hfill 505
:invert 475; 476; 477
:invertOval 475
:invertRect 475
:invertRoundRect 475
:paint 475; 476; 477
:paintOval 475
:paintRect 475
:paintRoundRect 475
:rb 500
:rb1 501
:row 504
:text 500
:userItem 503
:vfill 504
parasite 164
particular_value 105; 315
partie fermée d'une famille 179
pgcd 316
pile de backtracking (taille) 1009
pile pour la récupération de mémoire (taille) 1009
pixels 466; 471; 474; 482; 504; 511
plus unaire 98
police de caractères 469
ppcm 317
précision parfaite 26; 94
predefined 190; 318
préfixe 24
syntaxe 177; 434; 444
primitive de groupement 505
primitives
booléens 128
conversions de chaînes 86
numérique 104
tuples 80
primitives d'agencement 504
primitives de description 500
print_window 472
priorités
des opérateurs booléens 119
des opérateurs numériques 98
Pro3Final 1025; 1027
Pro3Goal 1025; 1026
Pro3Initial 1025
projection sur un sous-vocabulaire 127
prolog3.bst 1003
prolog3.era 1003
prolog3.erf 1004
prolog3.err 1004
prolog3.lex 1003
prolog3.prf 1004
prolog3.sya 1003
prolog3.syf 1003
prolog3.syn 1003
prologIII 319
protocole de transfert de données 1014; 1018
put 397

put_integer 1014; 1020; 1021
put_real 1014; 1020; 1021
put_string 1014; 1020; 1021
PutArg 1014; 1015

Q

qualifieur 24
quit 8; 198; 320; 1008

R

raccourci 496
rad 172; 370
radio-bouton 500
rafraîchissement de la fenêtre graphique 473
rational 106; 322
read 398
real 106; 323
récapitulatif
  opérations et relations booléennes 123
  opérations et relations définies sur les arbres et les tuples 79
redef_array 324
règles 60
  prédéfinies
    ajout de nouvelles 1010
    booléens 128
    numérique 104
    tuples 80
  syntaxe 440; 456
reinsert 188; 189; 325
relations 49
  booléennes 120
  comparaisons numériques 52
  de type 53
  égalité 50
  implication 52; 120
  inégalité 50
  nombre de fils 74
  numériques 100
  unaires 53
reload 191; 327
remove_implicit 183; 328
repeat 329
répertoire exemples 1004
requêtes 61
  syntaxe 440; 456
reset_cpu_time 198; 330
reset_window 472
résolution des systèmes de contraintes 57
restore_menubar 497
restrictions
  concaténation 77
  linéarité 102
  nombres entiers 102
  taille 76
resume_trace 331
retardements
  concaténation 154
  contraintes non-linéaires 106
  contraintes numériques 156
  freeze 149
  tailles 151
retour chariot 486; 490; 502; 511; 1008; 1010
retract 190; 332; 399
retractall 400
return 431
rubout 431
rule 190; 333
rule_nb 336

S

sauvegarde d'un module 11
sauvegarde de modules (save) 192
"SAVE" 468; 470
save 192; 337
save_menubar 497
save_window 469
see 401
seen 402
sémantique 62
  déclarative 62
  des termes 46
  opérationnelle 62
séparateur 177
set_config 108; 340
  echo 341

© PrologIA

epsilon_float 342
format_decimal 342
format_float 342
format_out_num 341
statistics 342
syntax 342
trace 340
tty_wrap 342
undefined_rule 343
set_configuration 344
set_context 180; 182; 345
set_ident 346; 1011
set_menu 493
set_window 470
sfgetfile 516; 517
sfputfile 517; 518
"SHAPE" 468
si_alors_sinon 166
simplification
contraintes booléennes 127
sin 172; 370
size 80; 152; 347
solution d'un système de contraintes 57
booléennes 126
sortie.log 1006
souris 480
soustraction 31; 98
split 81; 348
sqrt 173; 370
srcBic 483
srcCopy 483
srcOr 483
srcXor 483
state 349
string 87; 350
string_bool 86; 351
string_ident 86; 352
string_integer 86; 354
string_real 86; 355
stty 488
style 482; 483; 497
style_menu 497
sub 170; 368
substring 86; 356
sup 171; 369
supe 172; 369
supérieur ou égal 101
supérieur strict 101
suppress 10; 15; 191; 357
suppression
de messages 1007
de règles (suppress) 10; 191
suspend_trace 360
symboles relationnels
booléens 120
numériques 100
syntax 342
Edinburgh 342
Prolog III 342
syntaxes
choix du mode syntaxique 424
syntaxe de base 424
syntaxe Edimbourg 425
sys_command 361
systèmes de contraintes 54
syntaxe 438; 454

T

tab 403
tableaux (def_array) 174
tabulation 431; 486; 490; 504; 511
taille de la fonte 472; 482; 483; 484
taille de la police de caractères 468
tailles 76
retardées 151
tan 173; 370
tell 404
termes 35
connus 148
sans variable 37
syntaxe 435; 447; 450
tête de règle 60
told 405
trace 340; 362; 1008
trace_file 363
true 406
trunc 106; 172; 364; 369
"TTY" 467; 489
tty_wrap 342
tuple 81; 365

tuples 12; 22; 71; 76

U

undef_array 366
undefined_rule 343
  error 343
  fail 343
  warning 343
unités d'entrée/sortie 192
user_item 510
userItem 510
UserRule 1012; 1015; 1022
userrule.c 1011

V

val 169; 367
  abs 369
  add 368
  atan 370
  cos 370
  div 368
  eql 369
  exp 370
  if 369
  inf 369
  infe 369
  ln 370
  mod 369
  mul 368
  rad 370
  sin 370
  sqrt 370
  sub 368
  sup 369
  supe 369
  tan 370
  trunc 369
valeurs booléennes 25
var 407
variables 34
  d'écarts 104
  syntaxe 433; 443

W

"window" 497
write 408
writeq 409

© PrologIA

Association Prolog
HERITAGE

Prolog III

Version 1.3

Manuel de mise au point

Décembre 1991

© PrologIA

Association Prolog
HERITAGE

Garantie et responsabilités

PrologIA n'offre aucune garantie, expresse ou tacite, concernant ce manuel ou le logiciel qui y est décrit, ses qualités, ses performances ou sa capacité à satisfaire à quelque application que ce soit.

PrologIA ne pourra être tenue responsable des préjudices directs ou indirects, de quelque nature que ce soit, résultant d'une imperfection dans le programme ou le manuel, même si elle a été avisée de la possibilité que de tels préjudices se produisent. En particulier, elle ne pourra encourir aucune responsabilité du fait des données mémorisées ou exploitées, y compris pour les coûts de récupération ou de reproduction de ces données.

L'acheteur a toutefois droit à la garantie légale dans les cas et dans la mesure seulement où la garantie légale est applicable nonobstant toute exclusion ou limitation.

Droits d'auteur

Ce manuel et le logiciel qu'il décrit sont protégés par les droits d'auteur. Au terme de la législation traitant de ces droits, ce manuel et ce logiciel ne peuvent être copiés ou adaptés, en tout ou en partie, sans le consentement écrit de PrologIA, sauf dans le cadre d'une utilisation normale ou pour faire une copie de sauvegarde. Ces exceptions n'autorisent cependant pas la confection de copies à l'intention d'un tiers, que ce soit ou non pour les vendre.

Prolog III est une marque déposée de PrologIA.

Association Prolog
HERITAGE

Débuggeur et mise au point de programmes Prolog III

1. **Introduction et conseils**
2. **Trace d'exécution**
2. **Aperçu du modèle des boites**
3. **Des exemples**
4. **Règles prédéfinies de mise au point**
5. **Mise au point avancée**
6. **Détection d'appels non-définis**

Que trouve-t-on dans ce chapitre ?

Le propre de tout programme est de pouvoir être faux. Aucun langage de programmation ne permet en effet de rentrer en machine "l'algorithme que l'on a en tête" sans laisser la porte ouverte aux erreurs, qu'elles soient conceptuelles ou bien de simples fautes de frappe. Ce chapitre décrit les outils de mise au point mis à la disposition des programmeurs. On verra également que ces outils permettent plus que la simple correction d'erreur puisque le suivi d'exécution d'un programme permet d'améliorer, voire de saisir le comportement de l'algorithme programmé.

Novembre 1991

© PrologIA

1. Introduction

Les fautes qui ne provoquent pas d'erreur de syntaxe peuvent-être classées dans deux catégories :

- Celles qui sont détectables statiquement, c'est a dire en examinant l'ensemble des règles courantes que le programme est susceptible d'utiliser. Le cas le plus fréquent est l'appel à une règle non-définie. On utilisera l'outil *XRef* pour vérifier qu'un programme ne contient pas à un moment donné, de tels appels.
- Celles qui nécessitent l'exécution du programme

Le couple (identificateur, arité) identifie un paquet de règles

Faute de frappe dans un identificateur, faute d'inattention lorsqu'on appelle une règle formée d'un ident

Une des premières sources d'erreur est l'appel de règles non-définies (ou, du moins, pas encore définies au moment de l'appel dans le cas ou on utilise l'insertion dynamique de règles par les primitives assert, insert, reload). Ici, non-définies désigne aussi bien un 'bon' identificateur et une mauvaise arité.

Il faut se mettre en mode undefined_rule = warning ou error.

© PrologIA

2. Trace d'exécution

Le mode trace est un mode d'exécution permettant de visualiser le déroulement d'un programme Prolog III. On peut entrer et sortir de ce mode dynamiquement par des appels à des prédicats spécifiques[1]. Il est également possible de modifier les options de traçage en interrompant l'exécution en cours, en changeant les options en question, puis en poursuivant l'exécution du programme.

Ce traçage consiste essentiellement en une suite d'impressions, sur la sortie courante de trace, des contraintes d'égalité entre buts et tête de règles avant unification et de visualisation des points de backtracking. Ce listing contient des lignes de la forme :

```
but = tete_de_regle_a_essayer1          % indique une tentative
<<<<<<                                  % c'est un échec
but = tete_de_regle_a_essayer2          % une autre tentative
autre_but = autre_regle                 % l'effacement du but
                                        % précédent est un succès
```

La précision de la trace est paramétrable : on peut se placer dans un des trois modes d'impression des buts, selon la quantité d'information que l'on souhaite voir défiler à l'écran.

- mode 1 : impression des noms des prédicats appelés, sans leurs éventuels arguments.
- mode 2 : impression des appels avec arguments.
- mode 3 : impression des appels avec arguments et des contraintes portant sur les variables figurant dans le but et/ou la tête de règle à essayer.

Dans tous les cas, cette trace n'est pas interactive et ne permet qu'un suivi difficile de l'exécution d'un programme en raison notamment de la grande quantité d'informations affichée. Très souvent, on souhaite ne voir qu'une

[1] Voir *trace*, *no_trace*, *suspend_trace*, *resume_trace*, *set_config*, etc....

ou quelques portions restreintes de l'exécution. Bien qu'il soit possible d'utiliser les prédicats spécifiques[1] en des endroits stratégiques du programme, ceci ne peut se faire qu'on prix d'une réinsertion des règles modifiées et reste donc lourd d'emploi. Pour ces raisons, le débuggeur est préféré au simple traçage. Son interactivité permet de déterminer avec précision l'endroit ou l'on se trouve, de “parcourir” la démonstration en cours, d'imprimer des informations pertinentes, d'éviter les impressions superflues, de reprendre ou modifier l'exécution d'une section de programme, et bien d'autres choses encore.

3. Aperçu du modèle des boites

Le débuggeur Prolog III est basé sur le “modèle des boites” que l'on trouve dans la plupart des débuggeurs Prolog plus classiques. Le concept important de ce modèle est que **chaque appel peut être considéré comme une boite**, opaque ou transparente selon le désir de l'utilisateur d'avoir une trace d'exécution plus ou moins détaillée. Les boites de ce modèle comportent en général 4 ouvertures, appelées portes, mais un nombre de portes plus élevé est accessible à l'utilisateur Prolog III et sont décrites plus loin.

fig 1. Une boite opaque et ses quatre portes.

Ces portes correspondent à des endroits précis de l'exécution d'un but. Elles schématisent :

- l'activation d'un paquet de règles (CALL),
- la terminaison correcte d'une des règles du paquet (EXIT),
- le retour au sein de ce paquet par backtracking (REDO),
- la désactivation de ce paquet par épuisement de ses choix (FAIL).

1

Voici un exemple d'exécution, dans lequel ce qui est en **gras** doit être tapé par le programmeur. On insère deux petites règles, puis on se place en mode *débug* et on lance le but *toto(x);* . Le débuggeur attend une réponse à la fin de chacune des lignes affichant le prompt `(DBG)` . On se contentera pour l'instant de taper la lettre “s” signifiant “step” (on avance pas-à-pas) :

```
> toto(10)  ->;
> toto(11)  ->;
> debug;
{}
> toto(x);
  1[1]CALL    : toto(x) (DBG) s
  1[1]EXIT(r1): toto(10) (DBG) s
 0[0]EXIT(r1): $query$ (DBG) s
{x=10}
 0[0]REDO(r1): $query$ (DBG) s
  1[1]REDO(r1): toto(x) (DBG) s
  1[1]EXIT(r2*): toto(11) (DBG) s
 0[0]EXIT(r1): $query$ (DBG) s
{x=11}
>
```

fig 2.

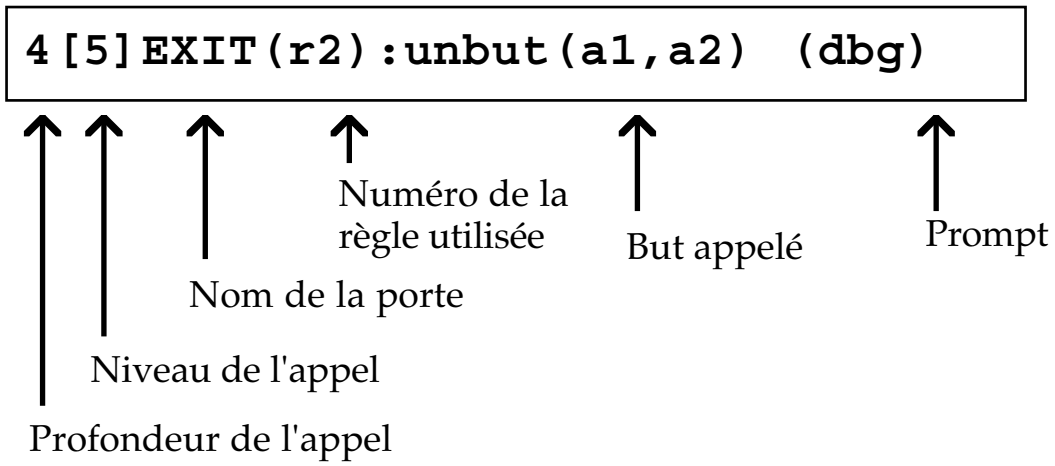
Dans le cas d'un appel à une règle prédéfinie on obtient la séquence :

```
> debug;
{}
> line;
  1[1]CALL    : line (DBG) s

  1[1]EXIT(rC): line (DBG) s
 0[0]EXIT(r1): $query$ (DBG) s
{}
>
```

fig 3.

Revenons à la figure 1. On constate que la plupart des lignes sont construites selon un même schéma. Voici la dénomination puis l'explication des différents champs de ce schéma :



Quelques définitions

Nom de la porte :

C'est une des chaînes CALL, RULE, {ok}, {no}, EXIT, REDO, FAIL .

Profondeur d'un appel (numéro de boite) :

C'est le numéro d'ordre de cet appel dans la branche courante de la démonstration. On peut dire que la profondeur d'un appel est son niveau dans l'arbre ET de la démonstration courante. La profondeur d'appel du premier but de la question[1] est 1. Ce numéro augmente en général d'une unité d'un appel au suivant, mais l'effacement de prédicats prédéfinis peut donner lieu à des variations supérieures à l'unité. Ce numéro est très important car il sert à identifier les boites vivantes de façon unique. On se rappellera que chaque boite correspond à un appel.

Profondeur de la démonstration :

C'est la profondeur de l'appel courant. Ce nombre augmente quand un appel est exécuté avec succès, et diminue lorsque l'appel échoue.

Niveau d'un appel :

[1] A moins que des contraintes retardées ne figurent dans la requête.

© PrologIA

Le niveau d'un appel correspond au niveau d'imbrication des boites et vérifie donc les deux conditions :

- Tous les buts de la question sont au niveau 1.
- Tous les sous-buts immédiats d'un corps de règle sont au même niveau, lequel est supérieur d'une unité au niveau de l'appel de cette règle.

4. Les commandes du débuggeur

1. Le langage de commande

Voici en quelques points la description du langage de commande du débuggeur Prolog III :

- Une commande est une suite de un ou plusieurs mots, terminée par une fin de ligne ou un point-virgule.
- Les mots de cette suite sont séparés par des espaces ou tabulations.
- Le premier mot est le nom de la commande, les mots suivants étant ses arguments.
- Les mots qui doivent contenir des blancs ou des tabulations doivent être parenthésés.
- Les noms de commandes doivent respecter les majuscules et minuscules.
- Par défaut, une ligne vide (ne contenant que des blancs et retour-chariot) signale l'exécution de la ligne précédemment fourni au débuggeur (on peut désactiver cette option nommée *nlrepeat* par la commande :

```
(DBG) set nlrepeat 0
```

- Un mécanisme d'alias permet de définir de nouvelles commandes en combinant des commandes plus élémentaires. Un certain nombre de commandes sont prédéfinies de cette façon.

- Les commandes sont exécutées à chaque point d'arrêt. S'il n'y a aucune commande en attente d'exécution, le prompt est affiché et l'utilisateur a la main.

2. Points d'impression et points d'arrêt

Par défaut, tout passage à une porte provoque l'impression de son statut. En mode pas-à-pas (*step*), ce point d'impression est aussi un point d'arrêt. Des commandes y seront lu et exécutées.

Il y a deux sortes de points d'arrêt, selon que l'on souhaite débugger un appel particulier ou tous les appels à un paquet de règles. Pour débugger la règle appelante, il faut utiliser la commande *breakat*, alors que pour débugger tous les appels à un paquet on utilise la commande *spy*. La différence entre ces commandes est que *breakat* met un point d'arrêt sur une boite de la démonstration courante alors que *spy* met un point d'arrêt sur l'ensemble des boites potentielles (qui existeront peut-être plus tard dans la démonstration).

3. Classifications des commandes

On peut classer les différentes commandes du débuggeur en plusieurs catégories :

- Les commandes de progression du démonstrateur Prolog III, utilisées pour faire des sauts plus ou moins grands :
  step, *next*, *nextp*, *nextc*, *cont*, *C*
- Les commandes utilisées pour imprimer des informations :
  chpt, *locus*, *ls*, *resol*, *rs*, *printbox*
- Les commandes qui installent ou suppriment des points d'arrêt :
  spy, *nospy*, *breakat*, *unbreak*
- Les commandes permettant de modifier l'exécution d'un programme :
  changechpt, *fail*, *FAIL*, *RULE*
- Les commandes qui gèrent les options du débuggeur :
  set, *show*, *p4*, *p7*, *t1*, *t2*, *t3*, *no_debug*, *kill*

© PrologIA

- Les commandes classées nulle-part ailleurs :
  alias, *eof*, *flush*, *help*, *source*

4. Progression

step, *next*, *n*, *N*, *nextp*, *nextc*, *cont*, *C*

step

tous ports

SYNTAXE :

```
step
```

DESCRIPTION :

C'est la commande de base pour suivre la progression de l'interprète. Elle permet de se déplacer à petit pas, de port en port. Seuls, les ports décrits dans l'option *printport* sont des points d'impression et points d'arrêt pour *step*. L'alias *s* est un raccourci de *step*.

REMARQUE :

Si l'option *print* est mise à 0, *step* ne fait qu'un grand pas, seulement interrompu par un éventuel point d'arrêt.

next, n, N

tous ports

SYNTAXE :

```
next
n
N
```

DESCRIPTION :

Cette commande permet la progression de l'interprète avec une granularité moins fine que *step*. En particulier *next* ne "rend la main" qu'une fois atteint un port qui n'est pas à un niveau supérieur au niveau courant. Les ports intermédiaires sont ou ne sont pas imprimés selon la valeur de l'option *print*.

Les points d'arrêt franchis par le déplacement sont actifs ou pas selon la valeur de l'option *break*. Seuls, les ports décrits par l'option *breakport* sont des points d'arrivée pour *next*. La commande *n* se comporte comme *next* à la différence qu'aucune impression des ports de boites intérieures n'a lieu. La commande *N* agit comme *n*, sauf que les point d'arrêts ne sont pas actifs. *N* et *n* sont des alias utilisant *next*.

---

cont, C

tous ports

SYNTAXE :

```
cont
C
```

DESCRIPTION :

Ces commandes sont généralement utilisées conjointement avec l'utilisation de points d'arrêt. *cont* "lâche la bride" à l'interpréteur, qui ne reviendra au mode interactif que lors de la rencontre d'un point d'arrêt. L'impression des ports a lieu ou pas selon la valeur des options *print* et *printport* . L'alias *C* se comporte comme *cont* à la différence qu'il n'y a pas d'impression, et rend les points d'arrêt actifs.

---

nextp

tous ports

SYNTAXE :

```
nextp
```

DESCRIPTION :

Cette variante de *next* permet de se déplacer jusqu'au prochain port de cette boite. Elle permet par exemple de passer du port `EXIT` au port `REDO`. Un cas particulier est de faire *nextp* alors qu'on sort irrémédiablement de la boite courante, que ce soit par le port `EXIT` sans `REDO` possible, soit par le port `FAIL`. Dans ces cas, l'option *warnnext* donne le comportement de *nextp*. Si l'option *warnnext* est à 1, un message d'erreur est imprimé et aucune action n'est accomplie. Si *warnnext* est à 0, un *step* est fait en lieu de *nextp*.

© PrologIA

5. Informations

On a la possibilité d'obtenir l'impression de diverses informations sur la démonstration courante au moyen des commandes *locus*, *ls*, *resol*, *rs*, et sur les différents choix en attente grâce à la commande *chpt*. la commande *sol* montre l'état du système de contraintes portant sur les variables de la requête. *printbox* sert à rafraîchir le statut du port courant.

---

chpt

tous ports

SYNTAXE :

`chpt`

DESCRIPTION :

Affiche les points de choix en attente, du plus récent au plus ancien (du haut vers le bas). L'impression est au format suivant :

```
--\/-- CHOICE POINTS (new)\/--
  5[2] (r3*): Dessert(glace,6)
  4[3] (r2*): Viande(porc,7)
  3[2] (r2*): Plat(p,i)
  2[2] (r2): HorsDoeuvre(pate,6)
--/\-- CHOICE POINTS (old)/\--
```

Les nombres affichés correspondent à ceux du format d'une ligne de description de port, c.a.d. la profondeur, le niveau entre [et], le numéro de règle à utiliser entre (et). Les termes imprimés correspondent aux têtes de règle en attente d'essai. Cet exemple indique que la règle n°3 (* indique que c'est la dernière) du paquet de nom *Dessert* sera essayée en premier.

REMARQUES :

Pour les prédicats prédéfinies, un point d'interrogation remplace les informations inconnues ou cachées.

---

locus

tous ports

SYNTAXE :

`locus`
`locus` *`nb_niveau`*

DESCRIPTION :

Affiche l'endroit le niveau courant de la démonstration sous la forme approximative d'une règle avec indication du niveau du corps de la règle et de la position de l'appel courant (boite courante) au sein de cette règle. Cette position est indiquée[1] par le caractère •. Quand un nombre est donné en argument, celui-ci désigne le nombre de niveau que l'on souhaite voir affiché, ligne par ligne, à partir du bas de l'arbre de démonstration (dans l'ordre des niveaux décroissants). Le niveau 0 correspond à l'appel virtuel du but *$query$* dont le corps de règle serait la requête tapée par l'utilisateur. Un point d'arrêt (pas les points-espions) est noté au moyen du caractère *, placé devant le littéral où il a été installé. Notons que l'interprétation de l'affichage du niveau affiché en premier dépend du port courant (le • peut être avant ou après un appel selon le port correspondant à cet appel).

```
> aa(x)  ->  bb(x)  cc(x);
> bb(x)  ->  eq(x,1);
> aa(u);

--------new query--------
   1[1]CALL    : aa(u) (DBG) s
   1[1]RULE(r1*): aa(u) = aa(x_1) (DBG)
   1[1]{ok}(r1*): aa(u) (DBG)
    2[2]CALL    : bb(u) (DBG)
    2[2]RULE(r1*): bb(u) = bb(x_2) (DBG)
    2[2]{ok}(r1*): bb(u) (DBG)
     3[3]CALL    : eq(u,1) (DBG)
      3[3]EXIT(r1*): eq(1,1) (DBG) locus
[3] bb(1) -> eq(1,1) •
 (DBG) locus 10
[3] bb(1) -> eq(1,1) •;
[2] aa(1) -> bb(1) •cc(1) ;
[1] aa(1) •;
[0]  $query$ •
 (DBG)
```

[1] Ce caractère dépend de l'implantation.

REMARQUE :

Si l'on n'était pas en mode débug au moment où la requête est lancée, *locus* ne peut donner d'information complète concernant la démonstration réalisée en amont du passage en mode débug.

---

ls

tous ports

SYNTAXE :

```
ls
```

DESCRIPTION :

Cette commande, implantée par un alias utilisant *locus*, est simplement un raccourci pour indiquer qu'on souhaite afficher la démonstration courante c.a.d. l'ensemble des appels depuis la requête.

---

resol

tous ports

SYNTAXE :

```
resol
resol      nb_resolvantes
```

DESCRIPTION :

Cette commande, assez similaire à la commande *locus*, permet de montrer les résolvantes locales (restes de corps de règles), par niveau, celles-ci correspondant aux littéraux qui restent à effacer. L'impression correspond en gros au buts situés à droite du caractère • visible par la commande *locus*. Avec un argument, `nb_resolvantes` correspond au nombre de niveaux à imprimer à partir du bas de l'arbre

```
> aa(x)  ->  bb(x)  cc(x);
> bb(x)  ->  eq(x,1);
> aa(u);

--------new query--------
   1[1]CALL    : aa(u) (DBG) s
   1[1]RULE(r1*): aa(u) = aa(x_1) (DBG)
   1[1]{ok}(r1*): aa(u) (DBG)
    2[2]CALL    : bb(u) (DBG)
    2[2]RULE(r1*): bb(u) = bb(x_2) (DBG)
    2[2]{ok}(r1*): bb(u) (DBG)
     3[3]CALL    : eq(u,1) (DBG) resol
eq(u,1) ;
 (DBG) resol 10
eq(u,1) ;
cc(u) ;
 (DBG)
```

rs

tous ports

SYNTAXE :

`rs`

DESCRIPTION :

Cette commande, implantée par un alias utilisant *resol*, est simplement un raccourci pour indiquer qu'on souhaite afficher toutes les résolvantes c.a.d. l'ensemble des littéraux en attente d'effacement.

sol

tous ports

SYNTAXE :

`sol`

DESCRIPTION :

Provoque l'affichage de l'état courant du système de contraintes portant sur les variables de la question. C'est le format utilisé lors de l'affichage des solutions Prolog III. Si la démonstration courante n'aboutit pas (on est sur `FAIL`, ou sur `{no}` sans autre choix possible), la chaîne "`none`" est imprimée en place du système, qui est dans ces cas-là insoluble.

© PrologIA

printbox

tous ports

SYNTAXE :

`printbox`

DESCRIPTION :

Cette commande permet d'imprimer la ligne formattée contenant les informations sur le port courant de la boite courante. L'alias *p* est un raccourci de *printbox*.

6. Installer des points d'arrêt

Les points-espions sont installés par *spy* et enlevés par *nospy*. Les points d'arrêts proprement dits, que l'on place dans la démonstration courante, sont installés par *breakat* et enlevés pas *unbreak*.

spy

tous ports

SYNTAXE :

`spy`
`spy` *`identificateur arité`*

DESCRIPTION :

Cette commande permet de surveiller tous les appels à un paquet de règle. Elle ne surveille donc pas un appel particulier comme le fait la commande *breakat*. Sans arguments, le nom et l'arité du paquet de règles correspondent à ceux de la boite courante. Avec arguments, décrit le paquet *`identificateur arité`*.

```
spy Dessert 2
```

Il est également possible d'ajouter des points-espions avec la primitive Prolog III *spy* .

nospy

tous ports

SYNTAXE :

`nospy`
`nospy` *`identificateur arité`*

DESCRIPTION :

Cette commande permet d'enlever tous les appels à un paquet de règle. Sans arguments, le nom et l'arité du paquet de règles correspondent à ceux de la boite courante. Avec arguments, décrit le paquet décrit par le couple *`identificateur arité`*.

```
nospy Dessert 2
```

Il est également possible d'enlever des points-espions avec la primitive Prolog III *no_spy* .

breakat

tous ports

SYNTAXE :

`breakat`
`breakat` *`n°_boite`*
`breakat` *`niveau`* *`n°_littéral`*

DESCRIPTION :

Cette commande permet de surveiller un appel de la démonstration courante en y mettant un point d'arrêt. Sans arguments, met un point d'arrêt sur la boite courante. Avec un argument, met un point d'arrêt sur la boite *`n°_boite`*. Avec deux arguments, décrit l'appel à surveiller en donnant "en coordonnées cartésiennes" sa localisation, dans l'affichage produit par la commande *ls* ou *locus* . Dans cet affichage, *`niveau`* décrit est le nombre indiqué en début de chaque ligne, et *`n°_littéral`* est le nombre de termes, comptés à partir de la gauche, la tête de règle étant comprise et en position 1.

© PrologIA

L'ensemble des noms de ports sur lequel le point d'arrêt s'arrêtera est la valeur de l'option *breakport* [1] au moment de l'installation du point d'arrêt.

Un point d'arrêt installé par *breakat* apparait sous la forme du caractère * dans l'affichage produit par la commande *ls* ou *locus* . Les points d'arrêt sont rendus inactifs en mettant l'option *break* à 0. Si l'option *break* est à 1, une commande de type *cont* ou *C* s'arrêtera sur ce point d'arrêt[2] pour les ports convenus lors de l'installation.

```
> aa(x)  ->  bb(x)  cc(x);
> bb(x)  ->  eq(x,1);
>  cc(1)  ->;
> aa(X);
--------new query--------
** 1[1]CALL    : aa(X) (DBG) s
** 1[1]RULE(r1*): aa(X) = aa(x_1) (DBG)
** 1[1]{ok}(r1*): aa(X) (DBG) ls
[2] aa(X) -> •bb(X) cc(X) ;
[1] aa(X) •;
[0]  $query$ •
 (DBG) breakat 2 3
 (DBG) ls
[2] aa(X) -> •bb(X) *cc(X) ;     Un point d'arrêt sur cc(X)
[1] aa(X) •;
[0]  $query$ •
 (DBG) C       On va jusqu'au prochain point d'arrêt
**    4[2]CALL    : cc(1) (DBG)  Point d'arrêt atteint
```

unbreak

tous ports

SYNTAXE :

unbreak
unbreak *n°_boite*
unbreak *niveau* *n°_littéral*

DESCRIPTION :

[1] Les options sont positionnée par la commande *set*,

[2] Si on y arrive, un backtracking pouvant très bien se produire entre temps.

Cette commande, de syntaxe similaire à *breakat*, permet d'enlever des point d'arrêts de la démonstration courante. Contrairement à *breakat*, tous les ports sont concernés, sans tenir compte de la valeur de l'option *breakport*.

7. Modifier le déroulement de l'exécution

Quelques commandes permettent de modifier le déroulement d'un programme; ce sont : *changechpt*, *fail*, *FAIL*, *RULE*.

RULE et *changechpt* permettent de modifier des points de choix.

FAIL et *fail* provoquent des backtracking.

---

changechpt

tous ports

SYNTAXE :

`changechpt` *`n°boite n°règle`*

DESCRIPTION :

Change le numéro de règle suivante du point de choix de la boite de numéro *`n°boite`* à la règle de numéro *`n°règle`*.

Cette commande permet de modifier un point de choix et modifie donc l'arbre de recherche, qui est parcouru par l'interpréteur. Appelons ici arbre de recherche l'arbre des démonstration dont les nœuds sont des 'OU' dont les fils correspondent aux alternatives au sein d'un paquet de règles. Cette commande ne modifie pas le parcours de la branche courante puisqu'elle modifie une alternative située dans le passé de cette branche.

changechpt permet d'élaguer certaines branches quand on sait qu'elles sont stériles, ou qu'on souhaite progresser plus vite dans la recherche d'une erreur. Il est également possible d'en réexécuter certaines qui ont déjà été parcourues. Cette commande utilise les informations affichées par la commande *chpt* .

Voici deux exemples où on modifie des choix en attente. Dans le celui-ci, on “saute” une solution :

```
> dizaine(10)  ->;
> dizaine(20)  ->;
> dizaine(30)  ->;
> dizaine(x);
--------new query--------
   1[1]CALL    : dizaine(x) (DBG) s
   1[1]EXIT(r1): dizaine(10) (DBG)
  0[0]EXIT(r1): $query$ (DBG) chpt
--\/ CHOICE POINTS (new)\/--
  1[1] (r2): dizaine(20)
__/\ CHOICE POINTS (old)/\__
 (DBG) changechpt 1 3
 (DBG) chpt
--\/ CHOICE POINTS (new)\/--
  1[1] (r3*): dizaine(30)
__/\ CHOICE POINTS (old)/\__
 (DBG) s
{x = 10}          et la prochaine solution sera 30
  0[0]REDO(r1): $query$ (DBG)
   1[1]REDO(r1): dizaine(x) (DBG)
   1[1]EXIT(r3*): dizaine(30) (DBG)
  0[0]EXIT(r1): $query$ (DBG)
{x = 30}
>
```

Dans celui-là, on “repasse” deux fois par la même solution :

```
> dizaine(x);
--------new query--------
   1[1]CALL    : dizaine(x) (DBG) s
   1[1]EXIT(r1): dizaine(10) (DBG)
  0[0]EXIT(r1): $query$ (DBG) chpt
--\/ CHOICE POINTS (new)\/--
  1[1] (r2): dizaine(20)
__/\ CHOICE POINTS (old)/\__
 (DBG) changechpt 1 1
 (DBG) chpt
--\/ CHOICE POINTS (new)\/--
  1[1] (r1): dizaine(10)
__/\ CHOICE POINTS (old)/\__
 (DBG) s
{x = 10}          et la prochaine solution sera 10
  0[0]REDO(r1): $query$ (DBG)
   1[1]REDO(r1): dizaine(x) (DBG)
   1[1]EXIT(r1): dizaine(10) (DBG)
  0[0]EXIT(r1): $query$ (DBG)
{x = 10}
...
{x = 20}
...
{x = 30}
>
```

REMARQUES :

- Le point de choix induit par le port `RULE` n'est réellement installé et exploitable avec les commandes *chpt* et *changechpt* que lorsque l'on passe sur le port `{ok}` , et ce, à condition que la création d'un point de choix soit justifiée (il n'y a pas de choix quand on exécute la dernière règle d'un paquet).
- On ne peut pas modifier un point de choix attaché à un prédicat prédéfini écrit en C (comme *enum*) .
- Un point de choix est susceptible de disparaître si un appel à *cut* ou au *garbage collector* se produit avant qu'on soit revenu à cette alternative.

fail

tous ports

SYNTAXE :

`fail`

© PrologIA

DESCRIPTION :

La commande *fail* fait échouer l'action en cours et dépend donc du port courant. On peut considérer cette commande comme l'exécution immédiate de la règle prédéfinie *fail*. Cette commande agit comme *step* lorsqu'on va en marche arrière (backtracking).

Le descriptif suivant indique les changements d'états, qui sont en fait assez intuitifs :

`CALL`	va en `FAIL` (sort de la boite)
`RULE`	va sur un autre `RULE` ou sur `FAIL`
`{ok}`	va sur `{no}`
`{no}`	fait *step* (va sur `RULE` ou `FAIL`)
`EXIT`	fait `REDO` ou `FAIL` (selon la présence ou pas d'un point de choix)
`REDO`	fait *step* (va sur `RULE` ou un autre `REDO` intérieur)
`FAIL`	fait *step* (va sur d'autres `FAIL` ou `REDO` extérieurs)

FAIL

tous ports

SYNTAXE :

`FAIL`

DESCRIPTION :

La commande *FAIL* (d'après le nom du port) fait immédiatement passer au port `FAIL` de la boite courante. Fait *step* dans le cas où on était déjà au port `FAIL` (c'est la poursuite du backtracking). Cette commande *FAIL* est plus "énergique" que la commande *fail* , qui est plus fine.

RULE

RULE, {no}, FAIL

SYNTAXE :

`RULE` *`n°règle`*

DESCRIPTION :

La commande *RULE* (d'après le nom du port) permet de sélectionner le numéro de la règle à utiliser, lorsqu'on est sur les ports `RULE` ou `{no}`. Cette commande est activable sur le port `FAIL` si celui-ci est consécutif à un port `{no}`.

```
> dizaine(x);
--------new query--------
   1[1]CALL    : dizaine(x) (DBG) s
   1[1]RULE(r1): dizaine(x) = dizaine(10) (DBG) RULE 3
   1[1]RULE(r3*): dizaine(x) = dizaine(30) (DBG) s
   1[1]{ok}(r3*): dizaine(30) (DBG)
   1[1]EXIT(r3*): dizaine(30) (DBG)
  0[0]EXIT(r1): $query$ (DBG)
{x = 30}
>
```

REMARQUES :

- Il y a bien sûr erreur lorsque le numéro fourni ne correspond à aucune règle au sein du paquet. Aucune action n'est alors faite.

8. Gestion des options du débuggeur

Ces options sont imprimables avec la commande *show*, et modifiables avec la commande *set* (décrites à la fin de cette section). Elle servent de paramétrage du débuggeur et en modifient le comportement, essentiellement au niveau de la quantité d'information affichée.

tracemode valeur : **1/2/3** valeur par défaut : **2**

Cette option indique quelle est la précision de l'impression des termes (voir trace) et prend donc une valeur parmi les nombre 1, 2 ou 3.

maxlevel valeurs : **≥ -1** valeur par défaut : **-1**

Indique quelle est le niveau maximal d'impression complète des arbres. Le niveau maximal d'un arbre étant 0 quand c'est une feuille, un entier plus grand sinon. Par exemple, si *maxlevel* vaut 0, le terme `eq(x, gg(y))` sera imprimé `eq(...,...)` et, si *maxlevel* vaut 1, il sera imprimé

© PrologIA

`eq(x, gg(...))` . La profondeur de cet arbre étant 2, positionner *maxlevel* à 2 l'imprime complètement. Une valeur de -1 indique une impression complète, sans restriction de niveau.

maxarg valeurs : **≥ -1** valeur par défaut : **-1**

Indique le nombre maximal d'arguments à imprimer, que ce soit des arguments de fonctions ou de tuple. Si *maxarg* vaut 0, le terme *eq(x, gg(y))* sera imprimé *eq(..(2)..))* indiquant que deux arguments sont présents et non-imprimés. Si *maxarg* vaut 1, *eq(x,..(1)..))* sera affiché, `..(n)..` indiquant qu'il reste une suite de *n* arguments non-imprimés. Une valeur de -1 indique une impression de tous les arguments.

maxexpr valeurs : **≥ -1** valeur par défaut : **-1**

Indique le nombre maximal de monômes à imprimer, au sein d'une expression numérique. Cette option se comporte un peu comme *maxarg*, et imprime le reste des monômes sous la forme `..(nm)..`, ou *n* est le nombre de monômes restant. Par exemple, si *maxexpr* vaut 1, le terme *eq(z+4y+(18/7)x-t, u)* sera imprimé *eq(z..(3m)..,u)* . Une valeur de -1 indique une impression de tous les monômes.

maxstr valeurs : **≥ -1** valeur par défaut : **-1**

Indique le nombre maximal de caractères à imprimer, au sein d'une chaîne de caractères (et non d'un tuple). Cette option se comporte comme maxexpr et imprime le reste de chaîne sous la forme `..(nc)..`, ou *n* est le nombre de caractères restant. Par exemple, si maxstr vaut 10, le terme *eq(x, "cette chaine est un peu trop longue")* sera imprimé : *eq(x,"cette chai..(25c)..")*. Une valeur de -1 indique une impression intégrale de la chaîne.

print valeurs : **0/1** valeur par défaut : **1**

Cette option pilote l'impression des ports pendant le déroulement d'un programme et affecte donc le comportement des commandes de progression, comme *step*, *next* et *cont* . En effet, tout point d'impression étant un point d'arrêt pour *step*, si cette option est mise à 0, *step* n'est plus en

mesure de s'arrêter[1]. Pendant l'exécution de *next*, aucun des ports correspondant à des boites intérieures n'est imprimé.

printport valeurs : **[cerfRYN]** valeur par défaut : **cerfRYN**

Cette option sert à limiter les impressions de ports ayant lieu pendant le déroulement du programme, que se soit par *step*, *next* ou *cont*. Cette option prend pour valeur un ensemble de ports, chacun décrit par une lettre, toutes les lettres collées. Il est possible de rajouter ou de retrancher des ports à cette option, en utilisant les signe + ou - . Quelques exemples :

```
set printport cerf
set printport +YN
set printport -RYN
```

Cette option affecte directement le comportement de *step*. Elle n'est exploitée que lorsque l'option *print* est mise à 1 .

break valeurs : **0/1** valeur par défaut : **1**

Cette option indique si on permet qu'un point d'arrêt, installé par les commandes *breakat* ou *spy*, provoque ou non une interruption du déroulement de la commande en cours. La commande *next* n'est pas affectée par cette option.

breakport valeurs : **[cerfRYN]** valeur par défaut : **cerfRYN**

Indique à quels ports la commande *next* peut arrêter sa progression. La syntaxe est du même type que pour l'option *printport*. L'autre usage de cette option concerne les point d'arrêts : la valeur de *breakport* au moment de l'installation d'un point d'arrêt par *breakat* indique à quels ports ce point d'arrêt peut "se réveiller". On rappelle que les points d'arrêts ne peuvent se réveiller que si l'option *break* est à 1 .

echo valeurs : **0/1** valeur par défaut : **0**

Imprime sur la sortie de trace les commandes qui sont exécutées.

prompt valeurs : **chaîne de 19 car. maxi** valeur par défaut : **(DBG)**

[1] A moins qu'un point d'arrêt n'ai été explicitement installé.

© PrologIA

Cette option contient le prompt utilisée par le débuggeur. La chaîne doit être parenthésée par (et) si elle contient des blancs ou point-virgules, comme par exemple :

```
set prompt ( (DBG) )
```

catcherror valeurs : **0/1/2** valeur par défaut : **2**

Positionnée à 1 ou 2, cette option indique si on souhaite localiser les erreurs ou les exécutions du prédicat prédéfini *block_exit* . Si l'option vaut 0, l'exécution de la session de débuggage ne sera pas interrompu par la détection d'erreur. Si l'option vaut 1, un message sera imprimé indiquant le type d'erreur (ou l'argument de *block_exit*) et le numéro de la boite où sera "récupérée" l'erreur le cas échéant. Si l'option vaut 2, le message précédent est imprimé et un port `EXIT` est affiché, laissant la main à l'utilisateur[1].

nlrepeat valeurs : **0/1** valeur par défaut : **1**

Quand cette option est positionnée à 1, indique si l'envoi d'une ligne vide (retour-chariot seulement) correspond en fait à l'envoi de la dernière ligne non-vide (contenant une ou plusieurs commandes). Très pratique quand on a une même suite de commande à faire, mais aussi très dangereux quand on a fait une suite de *fail* et qu'on a oublié de taper *step* ou *next* pour avancer ! Si l'option est à 0, l'envoi d'une ligne vide ne fait rien.

warnnext valeurs : **0/1** valeur par défaut : **0**

Quand l'option vaut 1, informe le cas échéant que les commandes *nextp* et *nextc* n'ont pu être exécutée normalement, généralement parce qu'un backtracking se produit, faisant sortir de la boite courante par le port `FAIL`.

option	valeurs	défaut	description
•print	0/1	1	-- Enable/disable Printing.
•printport	[cerfRYN]	cerfRYN	-- Print only specified ports (unless 'printat' has been used).
•break	0/1	1	-- Enable/disable Breaks (not Nexts).
•breakport	[cerfRYN]	cerfRYN	-- Break and Next only at these ports.

[1] Les commandes *fail* et *FAIL* ne sont pas active à ce port particulier.

- maxlevel >= -1 -1 (all) -- Max. level of full printing.
- maxarg >= -1 -1 (all) -- Max. number of arguments printed.
- maxexpr >= -1 -1 (all) -- Max. number of monomials printed.
- maxstr >= -1 -1 (all) -- Max. number of characters printed.
- tracemode 1/2/3 2 -- Printing mode for goals and terms.
- echo 0/1 0 -- Echo DBG commands before executing.
- prompt <19 chars ((DBG)) -- Dbg prompt string.
- catcherror 0..2 2 -- 0 = ignore, 1 = print error, 2 = print and catch (will print an EXIT port.)
- nlrepeat 0/1 1 -- if set, <CR> performs the previous command line.
- warnnext 0/1 0 -- Warns when nextc or nextp can't be done.

---

set

tous ports

SYNTAXE :

`set` *nom*
`set` *nom valeur*

DESCRIPTION :

Donne la valeur *valeur* à l'option *nom* parmi celles qui sont décrites plus haut. La forme à deux arguments positionne l'option à sa valeur par défaut.

---

show

tous ports

SYNTAXE :

`show`
`show` *nom*

DESCRIPTION :

La forme à un argument imprime la liste de toutes les options et leurs valeurs dans un format réutilisable. La forme à deux arguments permet de n'avoir que l'information concernant l'option *nom*.

---

no_debug

tous ports

© PrologIA

SYNTAXE :

`no_debug`

DESCRIPTION :

Cette commande sort du mode débug et poursuit l'exécution du programme en cours. Aucun point d'arrêt ou point-espion ne sera plus en mesure de se faire connaître (à moins que les prédicats prédéfinis *debug* ou *break* ne soient employés par le programme).

REMARQUE :

Il existe une primitive Prolog III qui s'appelle également *no_debug* .

---

kill

tous ports

SYNTAXE :

`kill`

DESCRIPTION :

Cette commande abandonne l'exécution du programme en cours et renvoie à l'interpréteur Prolog III qui attend un autre but. Le mode débug reste actif.

---

p4

tous ports

SYNTAXE :

`p4`

DESCRIPTION :

Cette commande positionne les options *breakport* et *printport*, de sorte que les seuls ports à prendre en compte soient les quatre ports “standard” `CALL`, `EXIT`, `REDO`, `FAIL`.

---

p7

tous ports

SYNTAXE :

```
p7
```

DESCRIPTION :

Cette commande positionne les options *breakport* et *printport*, de sorte que les tous les ports soient pris en compte. Cette commande est définie par un alias.

---

t1, t2, t3

tous ports

SYNTAXE :

```
t1
t2
t3
```

DESCRIPTION :

Cette commande positionne l'option *tracemode* en fonction du nombre qui apparaît dans la commande. On se référera à l'option *tracemode* et aux différents niveaux de trace pour plus de détails. Ces commandes sont définies par des alias.

9. Commandes annexes

Cette section décrit un certain nombre de commandes utilitaires. Certaines permettent de personnaliser son environnement sous débuggeur : *alias, eof, help, source* .

---

alias

tous ports

SYNTAXE :

```
alias
alias     nom
alias     nouveau_nom    (définition)
alias     nouveau_nom    (définition)    (aide)
```


© PrologIA

DESCRIPTION :

Permet de créer un synonyme d'un ensemble de commandes. La forme à un argument permet de lister l'ensemble des noms actuellement définis. La forme à deux arguments affiche la définition de l'alias de nom *nom* . La forme à trois arguments permet de définir *nouveau_nom* avec la définition *définition* qui est une suite d'une ou plusieurs commandes séparées par un point-virgule. La forme à quatre arguments est équivalente à la forme à trois, et permet en plus d'ajouter une ligne d'aide visible à travers la commande *help* .

L'intérêt d'un alias est de pouvoir donner un nom plus court (ou préféré) à la place du nom “officiel” d'une commande, ou bien de définir une nouvelle commande en combinant des commandes plus élémentaires.

EXEMPLES :

```
alias x next

alias nls (locus 10000;next)
```

REMARQUES :

- Les définitions ou aides contenant des blancs ou points-virgule doivent être parenthésées.
- Donner une définition vide au moyen de *()* efface un alias.
- Alias ne sert qu'à définir des commandes, ce n'est pas un mécanisme général de définition de macros.

---

eof

tous ports

SYNTAXE :

```
eof
eof  all
```

DESCRIPTION :

eof termine la lecture et l'exécution de commandes du débuggeur lues dans le fichier ou cette commande est présente. Avec un paramètre, ferme tous les fichiers de commandes actuellement ouverts.

REMARQUES :

eof ne ferme que les fichiers “disque”.

Voir la commande *source* pour lire un fichier de commandes.

---

help

tous ports

SYNTAXE :

```
help
help nom_de_commande
```

DESCRIPTION :

help permet simplement d'afficher la liste des commandes. Avec un argument, *help* imprime quelques informations sur `nom_de_commande` si celui-ci existe. Il est possible de documenter les alias (cf. *alias*).

---

source

tous ports

SYNTAXE :

```
source    nom_de_fichier
```

DESCRIPTION :

Cette commande permet d'ouvrir et de lire un fichier contenant des commandes du débuggeur Prolog III. Les commandes y sont lues, jusqu'à la fin du fichier, au fur et à mesure des besoins du débuggeur. Le fichier se substitue donc temporairement à la console. On utilise essentiellement cette commande pour le chargement d'alias et le positionnement de certaines options.

REMARQUES :

source est réentrant, et ce jusqu'à une profondeur de cinq appels.

La commande *eof* permet de terminer une lecture de fichier.

© PrologIA

5. Règles prédéfinies de mise au point

trace, no_trace, debug, no_debug, spy, no_spy, break

6. Mise au point avancée

1 Reprendre une démonstration (retry)

Il est très utile de pouvoir reprendre un démonstration sans avoir à la recommencer à partir du début, en particulier lorsque l'exécution du programme à pister est fastidieuse de par le temps d'exécution ou de la difficulté de se placer à ce point précis de la démonstration.

Généralement, lors de l'exécution d'un programme erroné, on a plus ou moins la certitude qu'une erreur se produit entre deux points P1 et P2 de cette démonstration, et le but de la session de débuggage est de réduire cet intervalle en parcourant l'intérieur par des sauts de plus en plus petits, afin de détailler l'exécution. Lors de cette recherche, souvent dichotomique, on passe de temps en temps “par dessus” l'erreur, car on retrouve la machine Prolog dans un état qu'on juge incorrect vis-à-vis du programme parfait qu'on a en tête. On a alors rapproché le nouveau point P2 de P1 et il n'y a plus qu'a débugger jusqu'à P2, si possible. Il est bien pratique d'éviter de relancer la question si on a la possibilité de rester sous la session courante, et de pouvoir revenir au point P1 afin de recommencer la section litigieuse du programme.

Cette première version du débuggeur ne permet pas d'effectuer une reprise automatique, mais fourni des moyens pour réaliser une “reprise manuelle” . Les commandes utiles à cette tâche sont décrites un peu plus haut, dans cette section; ce sont *changechpt*, *fail*, *FAIL*, *RULE* .

Les points de choix sont des jalons naturels de toute démonstration non-déterministe et sont les candidats tout désignés à la reprise d'exécution[1] . La manipulation consiste, pour refaire une portion de branche,

- à repérer le point de choix le plus proche possible du point erroné, en étant toutefois en amont (dans le passé) à l'aide de la commande *chpt*
- à effectuer une suite de *fail* ou *FAIL*, en retournant dans les boites pertinentes et se retrouver au port RULE correspondant à la boite du point de choix.
- à sélectionner la règle à utiliser[2] avec la commande *RULE*, afin de reprendre l'exécution.

La poursuite de l'exécution va refaire cette branche. Un exemple (il s'agit de "menu.p3") sera plus explicite :

```
> RepasLeger(h,p,d);
--------new query--------
   1[1]CALL    : RepasLeger(h,p,d) (DBG)
   1[1]RULE(r1*): RepasLeger(h,p,d) = RepasLeger(h_1,p_1,d_1) (DBG)
   1[1]{ok}(r1*): RepasLeger(h,p,d) (DBG)
    2[2]CALL    : HorsDoeuvre(h,i_1) (DBG) n
    2[2]EXIT(r1): HorsDoeuvre(radis,1) (DBG)
    3[2]CALL    : Plat(p,j_1) (DBG) n
    3[2]EXIT(r1): Plat(boeuf,5) (DBG)
```

Supposons qu'à ce moment nous souhaitions détailler l'exécution de la règle *Plat*, à partir du port CALL. On va chercher, parmi les points de choix, celui qui est avant la boite n°3 , laquelle correspond à *Plat* .

[1] Ce qui veut dire que les programmes déterministes ne permettent pas d'être repris.

[2] C'est souvent la précédente de celle qui est affichée.

© PrologIA

```
    3[2]EXIT(r1): Plat(boeuf,5) (DBG) chpt
--\/ CHOICE POINTS (new)\/--
  4[3] (r2*): Viande(porc,7)
  3[2] (r2*): Plat(p,i)
  2[2] (r2*): HorsDoeuvre(pate,6)
__/\ CHOICE POINTS (old)/\__
```

La boite nº 2 *HorsDoeuvre* possède un point de choix, il faut donc retourner jusque dans cette boite par une suite de commandes *FAIL* et *fail*.

```
    4[3]{ok}(r1): Viande(boeuf,5) (DBG) F
    4[3]FAIL(r1): Viande(p,j_1) (DBG)
   3[2]RULE(r2*): Plat(p,j_1) = Plat(p_3,i_3) (DBG)
   3[2]FAIL(r2*): Plat(p,j_1) (DBG)
   2[2]REDO(r1): HorsDoeuvre(h,i_1) (DBG) s
   2[2]RULE(r1): HorsDoeuvre(h,i_1) = HorsDoeuvre(radis,1) (DBG)
   2[2]{ok}(r1): HorsDoeuvre(radis,1) (DBG)
```

2. Récupérer une erreur

Il n'est pas toujours facile de déterminer l'origine d'une erreur d'exécution, quand bien même cette erreur a été récupérée par un mécanisme de blocs. La détection d'erreur se fait aisément puisque le positionnement de la seule option *catcherror* permet de mettre un point d'arrêt sur l'endroit ou cette erreur est générée.

```
> minimize(X);

--------new query--------
   1[1]CALL    : minimize(X) (DBG) c
** 1[1]: error (59) goes up to box 0[0]
   1[1]EXIT(r1): minimize(X_1) (DBG) c
Err 60: Expression non-bornee lors d'un calcul d'extremum
>
```

On se réfèrera à l'option *catcherror* pour plus d'information sur la capture d'erreurs. On notera également que le mécanisme général des blocs peut être pisté avec cette même méthode.

Il faut noter qu'à l'opposé des implantations existantes, notre modèle est complet et cohérent, représentant exactement une imbrication de boites aussi bien en descente qu'en remontée, avec en particulier un comportement symétrique des portes exit, redo et fail. Implanter ce modèle de façon complète a bien sûr amené un surcoût. Celui-ci est nul pour la porte call, faible pour les portes exit et fail, plus important pour la porte redo qui a été plus spécialement optimisée. Bien sûr, le fait que l'utilisateur soit susceptible de demander des informations sur le programme en cours, et en particulier des informations inutiles à l'interpréteur peut rendre une session de debugging prohibitif, puisque pouvant entraver la récupération de mémoire qui n'est alors plus totale et complète.

Les portes supplémentaires implémentées visent à déterminer plus finement le point d'exécution au sein de la boite et l'état du système de contraintes, avant ou après l'unification de la tête; il n'y a pas de surcoût particulier à ces portes.

ststs

© PrologIA

Ref. MPH-III/Fr/MMX

www.prolog-heritage.org